CE441: Data and Network Security
Defense Against the Control Hijacking Dark Arts

Behnam Momeni, PhD

Department of Computer Engineering
Sharif University of Technology

Fall 2019
Outline

1. Active Exploit Prevention
   - Stack Shield Defensive Measure
   - Stack Shield Countermeasures
   - Stack Guard Defensive Measure
   - Stack Guard Countermeasures

2. Passive Exploit Prevention

3. Moving Target Defense
Categories of Defensive Measures

- There are four types of defensive measures to stop control hijacking exploits
  1. To review software and fix their vulnerabilities
     - This is the most reliable solution
  2. Add extra codes to actively monitor execution and stop exploits
     - Stack Shield falls in this category
     - The extra overhead is inevitable
  3. Setup system in a special way to make exploits inexecutable
     - Passive nature of this category is beneficial performance-wise
  4. Moving target defense
     - Between two previous categories
     - There are some active operations to move the target program to a random location, but there is no active enforcement afterwards
     - Attacks will not be successful unless the randomness used is leaked somehow.
Categories of Defensive Measures

There are four types of defensive measures to stop control hijacking exploits:

1. To review software and fix their vulnerabilities
   - This is the most reliable solution

2. Add extra codes to actively monitor execution and stop exploits
   - Stack Shield falls in this category
   - The extra overhead is inevitable

3. Setup system in a special way to make exploits inexectutable
   - Passive nature of this category is beneficial performance-wise

4. Moving target defense
   - Between two previous categories
   - There are some active operations to move the target program to a random location, but there is no active enforcement afterwards
   - Attacks will not be successful unless the randomness used is leaked somehow.
Categories of Defensive Measures

- There are four types of defensive measures to stop **control hijacking** exploits
  1. To review software and **fix** their vulnerabilities
     - This is the most reliable solution
  2. **Add extra codes to actively monitor execution** and stop exploits
     - **Stack Shield** falls in this category
     - The extra overhead is inevitable
  3. **Setup system in a special way to make exploits **inexecutable**
     - Passive nature of this category is beneficial performance-wise
  4. **Moving target defense**
     - Between two previous categories
     - There are some active operations to move the target program to a random location, but there is no active enforcement afterwards
     - Attacks will not be successful unless the randomness used is **leaked somehow**.
Categories of Defensive Measures

There are four types of defensive measures to stop control hijacking exploits:

1. **To review software and fix their vulnerabilities**
   - This is the most reliable solution

2. **Add extra codes to actively monitor execution and stop exploits**
   - Stack Shield falls in this category
   - The extra overhead is inevitable

3. **Setup system in a special way to make exploits inexecutable**
   - Passive nature of this category is beneficial performance-wise

4. **Moving target defense**
   - Between two previous categories
   - There are some active operations to move the target program to a random location, but there is no active enforcement afterwards
   - Attacks will not be successful unless the randomness used is leaked somehow.
The **return address** and other **local variables** are stored in the same stack, hence, one could overwrite the other.

**Stack Shield** pushes a copy of the **return address** into a separate memory location (a **shadow stack** at the beginning of the **DATA** segment), hence, it cannot be overwritten by stack overflows.

Before returning, **two values** are compared and they must match to be used:
- or the non-overflowable **return address** can be used.
Outline

1. Active Exploit Prevention
   - Stack Shield Defensive Measure
   - Stack Shield Countermeasures
   - Stack Guard Defensive Measure
   - Stack Guard Countermeasures

2. Passive Exploit Prevention

3. Moving Target Defense
Stack Shield Countermeasures

- Stack smashing is only detected if the return address is modified.
- An adversary can:
  - overwrite return address with its original value
  - keep overwriting other values beyond the return address
  - and modify all local variables and function arguments
- This allows the program’s logic to be hijacked, for example, to login without knowing the username/password.
- What if there are function pointers or SEH records on the stack?
Stack Shield Countermeasures

- Stack smashing is only detected if the return address is modified.

- An adversary can:
  - overwrite return address with its original value
  - keep overwriting other values beyond the return address
  - and modify all local variables and function arguments

- This allows the program’s logic to be hijacked, for example, to login without knowing the username/password.

- What if there are function pointers or SEH records on the stack?
Evading /GS with exception handlers

- When exception is thrown, dispatcher walks up exception list until handler is found (else use default handler)

After overflow: handler points to attacker’s code exception triggered $\Rightarrow$ control hijack

Main point: exception is triggered before canary is checked

Borrowed from [40442-971:02-ctrl-hijacking.pdf], page 65
Defenses: SAFESEH and SEHOP

• /SAFESEH: linker flag
  – Linker produces a binary with a table of safe exception handlers
  – System will not jump to exception handler not on list

• /SEHOP: platform defense (since win vista SP1)
  – Observation: SEH attacks typically corrupt the “next” entry in SEH list.
  – SEHOP: add a dummy record at top of SEH list
  – When exception occurs, dispatcher walks up list and verifies dummy record is there. If not, terminates process.
Outline

1. Active Exploit Prevention
   - Stack Shield Defensive Measure
   - Stack Shield Countermeasures
   - Stack Guard Defensive Measure
   - Stack Guard Countermeasures

2. Passive Exploit Prevention

3. Moving Target Defense
Stack Guard Defensive Measure

- **Stack Canary** → a value which is not reproducible by adversary
- The **Stack Canary** is pushed after the sensitive values on the stack
- Overwriting the sensitive values using a buffer overflow vulnerability mandates overwriting the **Stack Canary** too
  - Since the adversary cannot reproduce the valid **canary**, checking its value before returning is used to inspect the stack sanity
  - Overflows are limited to local variables which are pushed after the **canary**
Building a Stack Canary – NULL Canary

- If a buffer is overflowed using functions like the strcpy, they will stop copying from the adversary-controlled input into the target buffer as soon as they reach to a NULL character.
- Hence, the NULL canary is not reproducible by strcpy.
- What about gets?
Building a Stack Canary – NULL Canary

- If a buffer is overflowed using functions like the `strcpy`, they will stop copying from the adversary-controlled input into the target buffer as soon as they reach to a NULL character.
- Hence, the NULL canary is not reproducible by `strcpy`.
- What about `gets`?
Building a Stack Canary – Terminator Canary

- The **NULL** character just terminates some unsafe functions
- Select a **canary** containing different terminating characters to stop more functions (*e.g. the* `gets` *reads until a* `newline` *character*)
- Candidate **terminator canary** → `0x000AFF0D`
- What if the unsafe function had no terminating/bad characters?
The **NULL** character just terminates some unsafe functions

Select a **canary** containing different terminating characters to stop more functions (*e.g.* the `gets` reads until a `newline character`)

Candidate **terminator canary** $\rightarrow 0\times00\text{AFF}0\text{D}$

What if the unsafe function had no **terminating/bad characters**?
Building a Stack Canary – Random Canary

- Select a random number
- If its entropy is high enough, adversary cannot guess it with acceptable probability
StackGuard (Cont.)

- StackGuard implemented as a GCC patch
  - Program must be recompiled

- Minimal performance effects: 8% for Apache

- Note: Canaries do not provide full protection
  - Some stack smashing attacks leave canaries unchanged

- Heap protection: PointGuard
  - Protects function pointers and setjmp buffers by encrypting them: e.g. XOR with random cookie
  - More noticeable performance effects

Dan Boneh

Borrowed from [40442-971:02-ctrl-hijacking.pdf], page 61
Outline

1. Active Exploit Prevention
   - Stack Shield Defensive Measure
   - Stack Shield Countermeasures
   - Stack Guard Defensive Measure
   - Stack Guard Countermeasures

2. Passive Exploit Prevention

3. Moving Target Defense
Stack Guard Countermeasures

1. Countermeasures which do not recover the canary
   - Stack smashing is not detected until returning from the function
   - So overwrite function local variables and arguments and use the function’s logic against itself
     - Read about other countermeasures in [four-tricks]

2. Countermeasures which discover the canary
int func(char *msg) {
    char buf[80];
    strcpy(buf, msg);
    process(buf);
    strcpy(msg, buf);
}
Stack Guard Countermeasures – Example

Arbitrary Write at Arbitrary Address

- Adversary can overflow `buf` by controlling `msg`
- So, the `return address` and the `msg` argument itself can be overwritten
  - Now, both of the `msg` pointer and the `buf` contents are in control of the adversary
  - Finally, contents of `buf` are copied into the area pointed by `msg`
    - A write arbitrary contents at arbitrary address primitive
- When returning, mismatch will be detected and other handling functions are called to terminate the program (e.g. a libc function for exiting)
  - Overwrite related `GOT` entries to invoke the shellcode instead of them
Adversary can overflow `buf` by controlling `msg`

So, the return address and the `msg` argument itself can be overwritten

Now, both of the `msg` pointer and the `buf` contents are in control of the adversary

Finally, contents of `buf` are copied into the area pointed by `msg`

  - A write arbitrary contents at arbitrary address primitive

When returning, mismatch will be detected and other handling functions are called to terminate the program (e.g. a libc function for exiting)

  - Overwrite related GOT entries to invoke the shellcode instead of them
Stack Guard Countermeasures – Example

Arbitrary Write at Arbitrary Address

- Adversary can overflow \texttt{buf} by controlling \texttt{msg}
- So, the \texttt{return address} and the \texttt{msg} argument itself can be overwritten
- Now, both of the \texttt{msg} pointer and the \texttt{buf} contents are in control of the adversary
- Finally, contents of \texttt{buf} are copied into the area pointed by \texttt{msg}
  - A \texttt{write arbitrary contents at arbitrary address} primitive
- When returning, mismatch will be detected and other handling functions are called to terminate the program (\textit{e.g. a libc function for exiting})
  - Overwrite related \texttt{GOT} entries to invoke the shellcode instead of them
Stack Guard Countermeasures

Guessing the Random Canary

- It is improbable to guess a high-quality random canary
- But it is probable to guess just one byte of such a canary
  - The probability is $\frac{1}{256}$
- If the random canary is not changed after each crash
  - When is this possible?
  - Adversary can find the first byte by 256 tries in the worst case
  - Knowing the correct value for the first byte, the second byte can be overwritten to find the value which causes no crash
  - This byte-by-byte method reveals the complete canary quickly
Stack Guard Countermeasures

Guessing the Random Canary

- It is improbable to guess a high-quality random canary.
- But it is probable to guess just one byte of such a canary.
  - The probability is $\frac{1}{256}$.
- If the random canary is not changed after each crash,
  - When is this possible?
  - Adversary can find the first byte by $256$ tries in the worst case.
  - Knowing the correct value for the first byte, the second byte can be overwritten to find the value which causes no crash.
  - This byte-by-byte method reveals the complete canary quickly.
StackGuard enhancements: ProPolice

- ProPolice (IBM) - gcc 3.4.1. (-fstack-protector)
  - Rearrange stack layout to prevent ptr overflow.

- String Growth
  - args
  - ret addr
  - SFP
  - CANARY
  - local string buffers
  - local non-buffer variables
  - copy of pointer args

  Protects pointer args and local pointers from a buffer overflow

Borrowed from [40442-971:02-ctrl-hijacking.pdf], page 62
What if can’t recompile: Libsafe

• **Solution 2**: Libsafe (Avaya Labs)
  – Dynamically loaded library (no need to recompile app.)
  – Intercepts calls to `strcpy (dest, src)`
    • Validates sufficient space in current stack frame:
      \[
      |\text{frame-pointer} - \text{dest}| > \text{strlen(src)}
      \]
    • If so, does `strcpy`. Otherwise, terminates application
Outline

1. Active Exploit Prevention

2. Passive Exploit Prevention
   - NX Defensive Measure
     - NX Countermeasures: ret2libc
     - NX Countermeasures: ROP

3. Moving Target Defense
NX Defensive Measure

- The shellcode was copied into a buffer to be executed
  - Buffer resided in the stack or heap
  - Ordinary codes reside in the .text section
- The exploit can be prevented by hardware support
  - OS marks different pages with a set of permissions
    - Pages which were intended to contain executable codes (e.g. .text) are given the execution permission
    - Pages which were intended to contain non-executable data (e.g. stack) are given the write permission
  - CPU inspects the page permission while converting a virtual address to a physical address and stops executing data or modifying codes mistakenly
NX Defensive Measure

- The **shellcode** was copied into a buffer to be executed
  - Buffer resided in the **stack** or **heap**
  - Ordinary codes reside in the **.text** section
- The exploit can be prevented by **hardware support**
  - **OS** marks different pages with a set of permissions
    - Pages which were intended to contain executable codes (e.g. **.text**) are given the **execution permission**
    - Pages which were intended to contain non-executable data (e.g. **stack**) are given the **write permission**
  - **CPU** inspects the page permission while converting a **virtual address** to a **physical address** and stops **executing data** or **modifying codes** mistakenly
Passive Exploit Prevention  NX Defensive Measure

**NX Defensive Measure**

- The **shellcode** was copied into a buffer to be executed
  - Buffer resided in the **stack** or **heap**
  - Ordinary codes reside in the **.text** section

**Some Names Used for This Technology**

- **NX**: No Execute
- **W ⊕ X**: Either **Write** or **Execute** but do not allow both
- **XD**: eXecute Disable
- **XN**: eXecute Never

address to a physical address and stops executing data or modifying codes mistakenly
NX Defensive Measure – Applicability

- Usually **NX** is harmless, hence, it is active by default
  - The `-z execstack` option in previous lecture was aimed to disable this feature
  - Run `readelf -l path/of/some/executable/file` to observe permissions of different program segments
- Some programs such as **Just In Time (JIT) compilers** have to dynamically generate executable codes
  - They can use the `mprotect` function to modify page permissions
  - If an adversary cannot execute codes without having an executable page, then it is safe to allow the running program to mark some dynamically prepared page as executable (and read-only) at runtime
  - So... *is it possible to execute codes without controlling an executable page?*
Usually **NX** is harmless, hence, it is active by default

- The `-z execstack` option in previous lecture was aimed to disable this feature
- Run `readelf -l path/of/some/executable/file` to observe permissions of different program segments

Some programs such as **Just In Time (JIT) compilers** have to dynamically generate executable codes

- They can use the `mprotect` function to modify page permissions
- If an adversary cannot execute codes without having an executable page, then it is safe to allow the running program to mark some dynamically prepared page as executable (and read-only) at runtime

So... *is it possible to execute codes without controlling an executable page?*
Outline

1. Active Exploit Prevention

2. Passive Exploit Prevention
   - NX Defensive Measure
   - NX Countermeasures: ret2libc
   - NX Countermeasures: ROP

3. Moving Target Defense
Program and its loaded libraries are executable themselves.

An adversary might find her desired logic by looking at the already existing codes instead of providing a custom shellcode.

To run `system("/bin/sh");...`

- `/bin/sh\0`: command to be executed
- `system()’s addr`: overwritten return address
- `0xdeadbeef`: fake ret-addr for `system()`
- `pointer to arg0`: first argument of `system()`
Program and its loaded libraries are executable themselves.

An adversary might find her desired logic by looking at the already existing codes instead of providing a custom shellcode.

To run `system("/bin/sh");...`

- `/bin/sh` command to be executed
- `system()`’s addr
- `0xdeadbeef` fake ret-addr for `system()`
- pointer to arg0 first argument of `system()`
Limitations of ret2libc

- What if a binary was not linked to libc.so at all?
  - Alternative linked libraries and the program itself can be searched for interesting functions

- What if the desired functionality was not implemented as a single function?
  - e.g. an adversary wants to connect back to her server and establish an interactive remote shell
  - Start by calling the first function (e.g. socket)
  - The return address of that function is in control of the adversary too
    - Return to the next required function (e.g. connect)

- What about 64-bit systems?
  - Arguments are passed in the registers, hence, they cannot be forged by overflowing the stack contents
Limitations of ret2libc

- What if a binary was not linked to libc.so at all?
  - Alternative linked libraries and the program itself can be searched for interesting functions
- What if the desired functionality was not implemented as a single function?
  - *e.g.* an adversary wants to connect back to her server and establish an interactive remote shell
    - Start by calling the first function (*e.g.* socket)
    - The return address of that function is in control of the adversary too
      - Return to the next required function (*e.g.* connect)
- What about 64-bit systems?
  - Arguments are passed in the registers, hence, they cannot be forged by overflowing the stack contents
Limitations of ret2libc

- What if a binary was not linked to libc.so at all?
  - Alternative linked libraries and the program itself can be searched for interesting functions

- What if the desired functionality was not implemented as a single function?
  - *e.g.* an adversary wants to connect back to her server and establish an interactive remote shell
  - Start by calling the first function (*e.g.* socket)
  - The return address of that function is in control of the adversary too
    - Return to the next required function (*e.g.* connect)

- What about 64-bit systems?
  - Arguments are passed in the registers, hence, they cannot be forged by overflowing the stack contents
Limitations of ret2libc

- What if a binary was not linked to libc.so at all?
  - Alternative linked libraries and the program itself can be searched for interesting functions

- What if the desired functionality was not implemented as a single function?
  - *e.g.* an adversary wants to connect back to her server and establish an interactive remote shell
  - Start by calling the first function (*e.g.* `socket`)
  - The return address of that function is in control of the adversary too
    - Return to the next required function (*e.g.* `connect`)

- What about 64-bit systems?
  - Arguments are passed in the registers, hence, they cannot be forged by overflowing the stack contents
Outline

1. Active Exploit Prevention

2. Passive Exploit Prevention
   - NX Defensive Measure
   - NX Countermeasures: ret2libc
   - NX Countermeasures: ROP

3. Moving Target Defense
What was especial about the functions in return to some function type of exploits?

- They execute some high-level functionality
- and return to an address which is given in the stack
- Sometimes, one function is enough
- And when it is not, more functions can be chained by their return addresses

It is not necessary to jump at the beginning of an existing function

- Jumping at any executable address which contains a series of instructions followed by a ret assembly instruction just works™!
  - Those instruction sequences are called gadgets
  - These are end-parts of existing functions
  - or unintended instructions which can be executed by jumping in the middle of intended instructions
What was especial about the functions in return to some function type of exploits?

- They execute some high-level functionality
- and return to an address which is given in the stack
- Sometimes, one function is enough
- And when it is not, more functions can be chained by their return addresses

It is not necessary to jump at the beginning of an existing function

- Jumping at any executable address which contains a series of instructions followed by a `ret` assembly instruction just works™!
  - Those instruction sequences are called gadgets
  - These are end-parts of existing functions
  - or unintended instructions which can be executed by jumping in the middle of intended instructions
Return Oriented Programming (ROP) Example

- ROP gadgets can be chained similar to ordinary functions by putting address of each gadget in place of the fake return address which will be used by the previous gadget.

```
; b8 89 41 08 c3  mov eax, 0xc3084189
; 89 41 08  mov [ecx+8], eax
; c3  ret

esp → ... 0x7c37638d  → pop ecx; ret
        0xF13C1A02  → pop edx; ret
        0x7c341591
        0xBAADF00D  → xor eax, eax; ret
        0x7c367042  → add eax, ecx; ret
        0x7c34779f  → mov ebx, eax; ret
        0x7c347f97
        ...  
```

Ref: [rop-turing]
Implementing Arbitrary Programs by ROP

- Find some gadgets for loading constants and/or memory contents into registers
- ...some gadgets to store registers into some memory addresses
- ...some gadgets for arithmetical operations between registers
- ...and some gadgets for conditional code execution
  - If RAX contains either 0 or 16, adding RAX to RSP will either changes nothing or skip two return addresses

- To search for ROP gadgets → Ropper
- A compiler to build ROP chains → EasyROP, ROPC
Implementing Arbitrary Programs by ROP

- Find some gadgets for loading constants and/or memory contents into registers
- ...some gadgets to store registers into some memory addresses
- ...some gadgets for arithmetical operations between registers
- ...and some gadgets for conditional code execution
  - If \texttt{RAX} contains either 0 or 16, adding \texttt{RAX} to \texttt{RSP} will either changes nothing or skip two return addresses
- To search for ROP gadgets $\rightarrow$ \texttt{Ropper}
- A compiler to build ROP chains $\rightarrow$ \texttt{EasyROP}, \texttt{ROPC}
Outline

1. Active Exploit Prevention
2. Passive Exploit Prevention
3. Moving Target Defense
   - ASLR Defensive Measure
     - ASLR Countermeasures: offset2lib
     - ASLR Countermeasures: Heap Spraying
Moving Target Defense

- Usually, the defending site is **static** and an adversary can **plan** the time and location of an attack
  - *e.g.* a bank’s safe in its central branch
- To eliminate this **advantage** of the adversary, defender may move the vulnerable target in an **unpredictable manner**
  - *e.g.* moving bank’s money between different safes (in different branches) in a way that nobody knows which location holds enough money to motivate a robbery
- This way, even knowing how to exploit a vulnerable target, the target might be somewhere else during the attack!
“The Conjuror,” painted by Hieronymus Bosch. The real trick of this painting is the pickpocket who is robbing the spectator who is bent over. Ref: https://upload.wikimedia.org/wikipedia/commons/2/2a/Conjurer_Bosch.jpg
User Space Memory Randomization

- User-space memory randomization defenses protect against memory-corruption attacks
  - Attackers require knowledge of the layout of memory
  - Defenses randomize layout
- E.g. Address Space Layout Randomization (ASLR)

Image Reference: Didier Stevens, yaisc.com

Borrowed from [effective-entropy:cset14-herlands.pdf], page 2
Static ASLR

- Stack, heap, and libraries are loaded at random addresses
- Executable itself is loaded at a fixed address
  - Program may depend on its loaded address
    - *e.g. due to branches to absolute addresses, global variable pointer arithmetic, etc.*
  - To randomize base address of an executable program, it should be compiled carefully to avoid bugs when moving its base address
    - One method is to keep symbols information in the executable and let the dynamic loader to assign them proper addresses → Relocatable Executable
    - Another method is to avoid using absolute addresses at all and compute all such addresses by adding an offset to the current instruction pointer → Position Independent Executable
- The advantage of PIE for shared libraries is that executable binary remains intact byte-by-byte after loading it at different virtual memory addresses, hence, it can be loaded once in the physical memory and be mapped to virtual memory of several processes
PIE ASLR – 32-bit

- All base addresses are randomized
- How much entropy does it provide?
  - In a 32-bit system, $32 - 12 = 20$ bits exist for pages
  - Stack may start anywhere near the end of the virtual address space (19 bits in Linux, 17 bits in Windows 8)
  - Heap is near the executable section and must not collide with the stack (13 bits entropy in Linux, 8 bits entropy in Windows)
  - Executable section itself has less randomness as it needs to allocate a larger continuous memory block (8 bits in both of Linux and Windows)
- Check [effective-entropy] for more details
PIE ASLR – 64-bit

- In a 64-bit system, larger address space allows more randomness to be provided
  - In Linux $\rightarrow 2^{28}$ bits (reported in [offset2lib])
  - In Windows $\rightarrow 2^{17}$ bits
- Aren’t those numbers too small for a 64-bit address space?
  - Indeed, there is a 48-bit address space currently and the 48th bit is repeated (sign-extended) to fill the 64-bit space
PIE ASLR – 64-bit

- In a **64-bit** system, larger address space allows more randomness to be provided
  - In Linux → $2^{28}$ bits (reported in [offset2lib](https://en.wikipedia.org/wiki/X86-64))
  - In Windows → $2^{17}$ bits

- Aren’t those numbers too small for a **64-bit** address space?
- Indeed, there is a **48-bit** address space currently and the 48th bit is repeated (sign-extended) to fill the 64-bit space

Outline

1. Active Exploit Prevention
2. Passive Exploit Prevention
3. Moving Target Defense
   - ASLR Defensive Measure
   - ASLR Countermeasures: offset2lib
   - ASLR Countermeasures: Heap Spraying
Using a stack buffer overflow vulnerability, it is possible to guess the return address similar to the stack canary.

After de-randomizing the .text section, address of loaded libraries will be de-randomized automatically as they are loaded at a constant offset relative to the .text section.

- Fixed in Linux kernel 4.1 by loading the PIE executable at a separate address (with a random offset to mmap base address).
Exploiting Linux and PaX ASLR’s weaknesses on 32- and 64-bit systems

Exploiting the Correlation weakness: offset2lib

With Linux Kernel < 4.1

```bash
# echo 2 > /proc/sys/kernel/randomize_va_space
# hello_world_dynamic_pie
7f621ffbb000-7f6220176000  r-xp 00000000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f6220176000-7f6220376000  ---p 001bb000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f6220376000-7f622037a000  r--p 001bb000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f622037a000-7f622037c000  rw-p 001bf000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f622037c000-7f6220381000  rw-p 00000000 00:00 0
7f6220381000-7f62203a4000  r-xp 00000000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f622059c000-7f622059d000  rw-p 00000000 00:00 0
7f622059d000-7f622059e000  r-xp 00000000 00:00 0
7f622059e000-7f62205a3000  rw-p 00000000 00:00 0
7f62205a3000-7f62205a4000  r--p 00220000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f62205a4000-7f62205a5000  rw-p 00230000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f62205a5000-7f62205a6000  rw-p 00000000 00:00 0
7f62205a6000-7f62205a7000  r-xp 00000000 00:02 4896 /bin/hello_world.dynamic_pie
7f62207a6000-7f62207a7000  r-xp 00000000 00:02 4896 /bin/hello_world.dynamic_pie
7f62207a7000-7f62207a8000  rw-p 00010000 00:02 4896 /bin/hello_world.dynamic_pie
7fff47e15000-7fff47e36000  rw-p 00000000 00:00 0 [stack]
7fff47e63000-7fff47e65000  r--p 00000000 00:00 0 [vvar]
7fff47e65000-7fff47e67000  r-xp 00000000 00:00 0 [vdso]
fffffffffffffff600000-fffffffffffffff601000  r-xp 00000000 00:00 0 [vsyscall]
```
Exploiting Linux and PaX ASLR’s weaknesses on 32- and 64-bit systems

Exploiting the Correlation weakness: offset2lib

With Linux Kernel $\geq 4.1$

```bash
# hello_world_dynamic_pie
54859ccd6000-54859ccd7000  r-xp  00000000  00:02  4896 /bin/hello_world_dynamic_pie
54859ced6000-54859ced7000  r--p  00000000  00:02  4896 /bin/hello_world_dynamic_pie
54859ced7000-54859ced8000  rw-p  00001000  00:02  4896 /bin/hello_world_dynamic_pie
7f75be764000-7f75be91f000  r-xp  00000000  00:02  5192 /lib/x86_64-linux-gnu/libc.so.6
7f75be91f000-7f75beb23000  ---p  001bb000  00:02  5192 /lib/x86_64-linux-gnu/libc.so.6
7f75beb23000-7f75beb25000  rw-p  001bf000  00:02  5192 /lib/x86_64-linux-gnu/libc.so.6
7f75beb25000-7f75beb2a000  rw-p  00000000  00:00  0  
7f75beb2a000-7f75beb4d000  r-xp  00000000  00:02  4917 /lib64/ld-linux-x86-64.so.2
7f75bed45000-7f75bed46000  rw-p  00000000  00:00  0  
7f75bed46000-7f75bed47000  r-xp  00000000  00:00  0  
7f75bed47000-7f75bed4c000  rw-p  00000000  00:00  0  
7f75bed4c000-7f75bed4d000  r-xp  00022000  00:02  4917 /lib64/ld-linux-x86-64.so.2
7f75bed4d000-7f75bed4e000  rw-p  00023000  00:02  4917 /lib64/ld-linux-x86-64.so.2
7f75bed4e000-7f75bed4f000  rw-p  00000000  00:00  0  
7fffbb3741000-7fffbb3762000  rw-p  00000000  00:00  0 [stack]
7fffbb377b000-7fffbb377d000  r-xp  00000000  00:00  0 [vvar]
7fffbb377d000-7fffbb377f000  r-xp  00000000  00:00  0 [vdso]
fffffffffffffff600000-fffffffffffffff601000  r-xp  00000000  00:00  0 [vsyscall]
```

Borrowed from [offset2lib:asia16-exploiting-aslr-weaknesses.pdf], page 52
1) Extracting static information

Memory map

<table>
<thead>
<tr>
<th>Address</th>
<th>Permissions</th>
<th>File Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>7fd1b414f000-7fd1b430a000</td>
<td>r-xp</td>
<td>/lib/.../libc-2.19.so</td>
</tr>
<tr>
<td>7fd1b430a000-7fd1b450a000</td>
<td>---p</td>
<td>/lib/.../libc-2.19.so</td>
</tr>
<tr>
<td>7fd1b450a000-7fd1b450e000</td>
<td>r-p</td>
<td>/lib/.../libc-2.19.so</td>
</tr>
<tr>
<td>7fd1b450e000-7fd1b4510000</td>
<td>rw-p</td>
<td>/lib/.../libc-2.19.so</td>
</tr>
<tr>
<td>7fd1b4510000-7fd1b4515000</td>
<td>rw-p</td>
<td></td>
</tr>
<tr>
<td>7fd1b4515000-7fd1b4538000</td>
<td>r-xp</td>
<td>/lib/.../ld-2.19.so</td>
</tr>
<tr>
<td>7fd1b4718000-7fd1b471b000</td>
<td>rw-p</td>
<td></td>
</tr>
<tr>
<td>7fd1b4734000-7fd1b4737000</td>
<td>rw-p</td>
<td></td>
</tr>
<tr>
<td>7fd1b4737000-7fd1b4738000</td>
<td>r-xp</td>
<td>/lib/.../ld-2.19.so</td>
</tr>
<tr>
<td>7fd1b4738000-7fd1b4739000</td>
<td>rw-p</td>
<td>/lib/.../ld-2.19.so</td>
</tr>
<tr>
<td>7fd1b4739000-7fd1b473a000</td>
<td>rw-p</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Permissions</th>
<th>File Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>7fd1b473a000-7fd1b473c000</td>
<td>r-xp</td>
<td>/root/server.64.PIE</td>
</tr>
<tr>
<td>7fd1b493b000-7fd1b493c000</td>
<td>r--p</td>
<td>/root/server.64.PIE</td>
</tr>
<tr>
<td>7fd1b493c000-7fd1b493d000</td>
<td>rw-p</td>
<td>/root/server.64.PIE</td>
</tr>
<tr>
<td>7fff981fa000-7fff9821b000</td>
<td>rw-p</td>
<td>[stack]</td>
</tr>
<tr>
<td>7fff983fe000-7fff98400000</td>
<td>r-xp</td>
<td>[vdso]</td>
</tr>
</tbody>
</table>

STACK

```
...  
BUFFER
RBP
0x??????????????????
...  
```

This value (0x00007F) can be obtained:

1. Running the application and showing the memory map.
2. Checking the source code if set any limit to stack.
1) Extracting static information

Memory map

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Permissions</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>7fd1b414f000-7fd1b430a000</td>
<td>r-xp</td>
<td>/lib/.../libc-2.19.so</td>
</tr>
<tr>
<td>7fd1b430a000-7fd1b450a000</td>
<td>---p</td>
<td>/lib/.../libc-2.19.so</td>
</tr>
<tr>
<td>7fd1b450a000-7fd1b450e000</td>
<td>r--p</td>
<td>/lib/.../libc-2.19.so</td>
</tr>
<tr>
<td>7fd1b450e000-7fd1b4510000</td>
<td>rw-p</td>
<td>/lib/.../libc-2.19.so</td>
</tr>
<tr>
<td>7fd1b4510000-7fd1b4515000</td>
<td>rw-p</td>
<td>/lib/.../ld-2.19.so</td>
</tr>
<tr>
<td>7fd1b4515000-7fd1b4538000</td>
<td>r-xp</td>
<td>/lib/.../ld-2.19.so</td>
</tr>
<tr>
<td>7fd1b4718000-7fd1b471b000</td>
<td>rw-p</td>
<td>/lib/.../ld-2.19.so</td>
</tr>
<tr>
<td>7fd1b4734000-7fd1b4737000</td>
<td>rw-p</td>
<td>/root/server.64_PIE</td>
</tr>
<tr>
<td>7fd1b4737000-7fd1b4738000</td>
<td>r--p</td>
<td>/root/server.64_PIE</td>
</tr>
<tr>
<td>7fd1b4738000-7fd1b4739000</td>
<td>rw-p</td>
<td>/root/server.64_PIE</td>
</tr>
<tr>
<td>7fd1b4739000-7fd1b473a000</td>
<td>rw-p</td>
<td>/root/server.64_PIE</td>
</tr>
</tbody>
</table>

STACK

- ...                      
- BUFFER
- RBP
- 0x00007F?????????????
- ...                      

This value (0x00007F) can be obtained:
1. Running the application and showing the memory map.
2. Checking the source code if set any limit to stack.

Borrowed from [offset2lib:asia16-exploiting-aslr-weaknesses.pdf], page 31
1) Extracting static information

Since the executable has to be `PAGE_SIZE` aligned, the 12 lower bits will not change when the executable is randomly loaded.

**ASM Code**

```
0000000000001063 <attend_client>:
1063: 55    push %rbp
1064: 48 89 e5  mov %rsp,%rbp
1067: 48 81 ec 60 04 00 00 sub $0x460,%rsp
106e: 64 48 8b 04 25 28 00 mov %fs:0x28,%rax
1075: 00 00
1077: 48 89 45 f8 mov %rax,-0x8(%rbp)
107b: 31 c0 xor %eax,%eax
..... ..... ......
12d7: 48 89 c7 mov %rax,%rdi
12da: e8 1c fc ff ff callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff lea -0x440(%rbp),%rax
12e5: 48 89 c7 mov %rax,%rdi
..... ..... [From the ELF] ..... 
```

**STACK**

```
[From the ELF]
```

Borrowed from [offset2lib:asia16-exploiting-aslr-weaknesses.pdf], page 32
1) Extracting static information

Since the executable has to be `PAGE_SIZE` aligned, the 12 lower bits will not change when the executable is randomly loaded.

ASM Code

```
0000000000001063 <attend_client>:
1063:  55  push %rbp
1064:  48 89 e5  mov %rsp,%rbp
1067:  48 81 ec 60 04 00 00  sub $0x460,%rsp
106e:  48 8b 04 25 28 00  mov %fs:0x28,%rax
1075:  00 00
1077:  48 89 45 f8  mov %rax,-0x8(%rbp)
107b:  31 c0  xor %eax,%eax
.....   .....   .....
12d7:  48 89 c7  mov %rax,%rdi
12da:  e8 1c fc ff ff  callq efb <vuln_func>
12df:  48 8d 85 c0 fb ff ff  lea -0x440(%rbp),%rax
12e3:  48 89 c7  mov %rax,%rdi
.....   .....   [From the ELF]  .....  
```

STACK

```
.....
  BUFFER
    RBP
  0x0007F?????????2DF
.....
```

Borrowed from [offset2lib:asia16-exploiting-aslr-weaknesses.pdf], page 33
Exploiting Linux and PaX ASLR’s weaknesses on 32- and 64-bit systems

Exploiting the Correlation weakness: offset2lib

2) Brute forcing Saved-IP address

```c
void vuln_func(char *str, int lstr){
    char buff[48];
    int i = 0;
    ...
    for (i = 0; i < lstr; i++) {
        if (str[i] != '\n')
            buff[lbuff++] = str[i];
    }
}
```

- The unknown 28 random bits: “byte-for-byte” attack.
- The first byte is “special”, we know the lowest 4 bits:
  - $0x?_{16} \rightarrow ??_{10} \rightarrow 2^4 = 16$ attempts
  - $\{0x02, 0x12, 0x22 \ldots 0xC2, 0xD2, 0xE2, 0xF2\}$
2) Brute forcing Saved-IP address

```c
void vuln_func(char *str, int lstr){
    char buff[48];
    int i = 0;
    ...
    for (i = 0; i < lstr; i++) {
        if (str[i] != \n')
            buff[lbuff++] = str[i];
    }
    ...
}
```

- The unknown 28 random bits: “byte-for-byte” attack.
- The first byte is “special”, we know the lowest 4 bits:
  - \(0x?2_{16} \rightarrow ??10_2 \rightarrow 2^4 = 16\) attempts
  - \{0x02, 0x12, 0x22 ... 0xC2, 0xD2, 0xE2, 0xF2\}

Borrowed from [offset2lib:asia16-exploiting-aslr-weaknesses.pdf], page 35
2) Brute forcing Saved-IP address

```c
void vuln_func(char *str, int lstr){
    char buff[48];
    int i = 0;
    ...
    for (i = 0; i < lstr; i++) {
        if (str[i] != '\n')
            buff[lbuff++] = str[i];
    }
    ...
}
```

- The unknown 28 random bits: “byte-for-byte” attack.
- The first byte is “special”, we know the lowest 4 bits:
  - $0x?2_{16} \rightarrow ??10_{2} \rightarrow 2^4 = 16$ attempts
  - $\{0x02, 0x12, 0x22 \ldots 0xC2, 0xD2, 0xE2, 0xF2\}$
- The remaining 3 bytes $\rightarrow$ standard “byte-for-byte” attack
  - $3 \times 2^8 = 768$ attempts.
- After execute the byte-for-byte we obtained $0x36C6FE$
2) Brute forcing Saved-IP address

```c
void vuln_func(char *str, int lstr)
{
    char buff[48];
    int i = 0;
    ...
    for (i = 0; i < lstr; i++) {
        if (str[i] != '
')
            buff[libuff++] = str[i];
    }
    ...
}
```

- The unknown 28 random bits: “byte-for-byte” attack.
- The first byte is “special”, we know the lowest 4 bits:
  - \(0x?2_{16} \rightarrow ??10_2 \rightarrow 2^4 = 16\) attempts
  - \(\{0x02, 0x12, 0x22, \ldots, 0xC2, 0xD2, 0xE2, 0xF2\}\)
- The remaining 3 bytes \(\rightarrow\) standard “byte-for-byte” attack
  - \(3 \times 2^8 = 768\) attempts.
- After execute the byte-for-byte we obtained \(0x36C6FE\).

Borrowed from [offset2lib:asia16-exploiting-aslr-weaknesses.pdf], page 37
2) Brute forcing Saved-IP address

```c
void vuln_func(char *str, int lstr)
{
    char buff[48];
    int i = 0;
    ...
    for (i = 0; i < lstr; i++) {
        if (str[i] != '\n')
            buff[lbuff++] = str[i];
    }
    ...
}
```

- The unknown 28 random bits: “byte-for-byte” attack.
- The first byte is “special”, we know the lowest 4 bits:
  - $0x?2_{16} \rightarrow ??10_2 \rightarrow 2^4 = 16$ attempts
  - $\{0x02, 0x12, 0x22, \ldots, 0xCF\}$
- The remaining 3 bytes $\rightarrow$ standard “byte-for-byte” attack
  - $3 \times 2^8 = 768$ attempts.
- After execute the byte-for-byte we obtained $0x36C6FE$
- We need to perform $\frac{2^4 \times 3 \times 2^8}{2} = 392$ attempts on average.

Borrowed from [offset2lib:asia16-exploiting-aslr-weaknesses.pdf], page 38
5) Getting app. process mapping

Obtaining library base addresses:
- Application Base = 0x7FD1B473A000
- Offset2lib (libc) = 0x5eb000
- Offset2lib (ld) = 0x225000

Libc Base = 0x7FD1B473A000 - 0x5eb000 = 0x7FD1B414F000
Id Base = 0x7FD1B473A000 - 0x225000 = 0x7fd1b4515000

Borrowed from [offset2lib:asia16-exploiting-aslr-weaknesses.pdf], page 46
Outline

1. Active Exploit Prevention
2. Passive Exploit Prevention
3. Moving Target Defense
   - ASLR Defensive Measure
   - ASLR Countermeasures: offset2lib
   - ASLR Countermeasures: Heap Spraying
Allocating $x$ bytes in the heap

- will return a pointer to a random address
- which depends on the heap base address
- and the previously allocated parts of the heap
  
  *e.g.* if $x$ bytes were allocated and released previously (creating a hole in the heap), the same memory pointer will be returned for the new allocation

Allocating $x$ bytes repeatedly, will fill all corresponding heap holes and will continue towards higher addresses

Repeating the allocation enough times, it will cover up the variation at the heap base address and will eventually reaches to an arbitrary higher address

* e.g. put the desired payload at $0x0C0C0C0C$
Heap Spray – Memory Layout

Heap Spray – Usages

- What is the benefit of loading payloads at known locations?
- If a function pointer (or SEH record) can be controlled to jump at one specific address
  - Executable Heap $\rightarrow$ put shellcode at some known address and jump there
  - Non-Executable Heap $\rightarrow$ put ROP chain at some address which is pointed by some variable (register or stack living)
    - ...and jump into a stack pivoting gadget to begin the ROP chain execution


References and Further Reading (2/2)
