CS 177 - Computer Security

Memory Corruption
Agenda

• Intel x86-64 (64-bit) architecture basics
• Stack-based overflows
  – A deeper look into the stack
  – Taking control of the program
  – The shellcode
• Defenses and evolution of attacks
• Other overflows and considerations
X86 (64-BIT) ARCHITECTURE AND ASSEMBLER BASICS
Computer Architecture

• The modern computer architecture is based on “Von Neuman”
  – Two main parts: CPU (Central Processing Unit) and Memory
  – This architecture is used everywhere (incl., cell phones)
  – This architecture is fundamental, has not changed

• What is memory used for?
  – E.g., location of curser, size of windows, shape of each letter being displayed, graphics of icons, text, values, etc.
  – Von Neuman also says that not only data, but programs (code) should be in memory, too
The CPU

• Storing data by itself, of course, is not enough
  – CPU reads instructions from memory (one after the other)
  – Then executes each instruction (fetch-execute-cycle)

• Some important components that make up the CPU
  – General-purpose registers
  – Arithmetic and logic unit (ALU)
  – Special registers, incl. program counter
This Class

- Focus on Intel x86 architecture (64-bit architecture)
- Illustrates principles well
  - similar problems appear on other architectures
Intel x86 Architecture

• Very popular, but “crazy”
• CISC (complex instruction set computing)
  – hundreds of distinct opcodes
• Variable-length instructions
• Built of many backwards-compatible revisions
• Register-poor
  – 32-bit architecture has only six general purpose registers
  – 64-bit architecture extends this to 16 general registers (R0..R15)
  – R0..R7 have aliases RAX, RCX, RBX, RDX, RSP, RBP, RSI, RDI
x86-64 Registers

- General purpose registers
  - for “normal” computation, to store addresses, ...
- Floating point registers (FPU)
- XMM registers
- Segment registers
  - for segmentation-based memory access
- Special purpose registers:
  - Instruction pointer (RIP)
  - Flags register (RFLAGS)
  - Stack pointer (RSP)
  - Base pointer (RBP) – can be used as a general register as well
Program Counter (RIP)

- Is used to tell the CPU where to fetch next instruction
  - there is no difference between memory and data
  - program counter holds memory address of next instruction

- Instruction decoder then makes sense of the instruction
  - Instructions often include memory locations as well
    - move this piece of data from address X to address Y in memory
Important Instructions

• Data move instruction
  – mov: used often to move around data

• Arithmetic and logic instructions
  – add, sub, mul, and, or, xor, ...

• Stack manipulation
  – push, pop

• Control flow instructions
  – compare instruction: cmp
  – branch and jump instructions: je, jg, jge, jl, jle, jmp
Data Accessing Methods

Many different ways of accessing data in memory

- **Immediate mode**
  - Value is part of instruction itself

- **Register addressing mode**
  - Instruction references a register (rather than memory location)

- **Direct addressing mode**
  - Instruction references a memory address (that is accessed)

- **Indirect addressing mode**
  - Instruction references a register that holds the memory address (that is accessed)
Instruction Syntax (AT&T Syntax)

- **Format**: `opcode src, dst`
- **Constants preceded by $**: `mov $16, %rbx`
- **Registers preceded by %**: `mov %rax, %rbx`
- **Indirection uses ( )**: `mov (%rax), %rbx`
  
  `mov 4(%rax), %rbx`
  
  The item stored at `%rax + 4`
Process Memory Layout

- **.text**
  - machine code of executable
- **.data**
  - global initialized variables
- **.bss**
  - global uninitialized variables

### Example: Outside of any function:

```
int val = 3;
char string[] = "Hello World";
```

### Example: Outside of any function:

```
static int i;
```
Process Memory Layout

- **.text**
  - machine code of executable

- **.data**
  - global initialized variables

- **.bss**
  - global uninitialized variables

- **heap**
  - dynamic variables (malloc)

- **stack**
  - local variables and function call information (frames)

- **env**
  - environment variables and arguments
• Process memory layout is entirely virtual, and for now, we do not need to understand how it really works!

• Process memory layout thus always addresses the whole (in our case, 64-bit) address space.
sub Instruction

- Subtract from a register value

```
sub %rax, %rbx
```
push Instruction

- Put a value on the stack
  - value from register
  - value goes to (%rsp)
  - subtract 8 from %rsp
- Example
  push %rax
pop Instruction

- Take value from the stack
  - value from (%rsp)
  - value goes into register
  - add 8 to %rsp
- Example
  pop %rax
jmp Instruction

• Control flow transfer
  – %rip points to the currently executing instruction (in the text section)
  – Has unconditional and conditional forms
  – Example uses relative addressing

%rip K %rbp M %rsp N

K: jmp $-20

Stack

%rip K-20 %rbp M %rsp N

Stack

%rip K-20 %rbp M %rsp N

Stack
call Instruction

- Used for function calls
- Saves the current instruction pointer to the stack
- Jumps to the argument value
ret Instruction

- Return from a function call
- Pops the top value of the stack into the instruction pointer
leave Instruction

• Prepare return
• Equivalent to
  mov %rbp, %rsp
  pop %rbp

K: leave
K+i: ret

%rip K %rbp M %rsp N

Stack

%rip K+i %rbp A %rsp M

Stack

ret
BUFFER OVERFLOWS
Buffer Overflows

- A buffer overflow occurs any time the program attempts to store data beyond the boundaries of a buffer, overwriting the adjacent memory locations

- Goals
  - Overwrite other “interesting” variables (file names, passwords, pointers...)
  - Force program to execute operations it was not intended to do
    - inject (or simply find) code into the process memory
    - change flow of control (flow of execution) to execute that code
Buffer Overflows

- Common targets
  - setuid/setgid programs
  - network servers

- Vulnerable software
  - Mostly C/C++ programs
  - Programs written in memory-safe languages (Java, Python, C#) are typically safe
Buffer Overflows

• Stack-based buffer overflows are the quintessential memory corruption vulnerability
  – problem known since the 1970s, but first exploited by the Morris worm in 1988
  – rediscovered in a 1995 Bugtraq post
  – Aleph One wrote an accessible Phrack article in 1996
  – people suddenly realized they were everywhere...
Part I
A deeper look into the stack
The Stack

• In most architectures (Intel, Motorola, Sparc), stack grows towards bottom

• A running program uses the stack to enable functions to work properly

• For each function that is invoked at runtime, we allocate a (stack) frame for this function

• Each frame stores a number of important pieces of data for this function
Stack Frames

- A stack frame can be used to hold
  - (some) function parameters
    - items passed to function for processing
    - in x86-64, first six arguments are passed in registers (in order: RDI, RSI, RDX, RCX, R8, R9)
  - local variables
    - temporary storage areas used in the function
    - thrown away when the processing finishes
  - return address
    - where to jump when you are done
    - when a function is invoked, the calling point is saved
    - when the function completes, it returns to the initial calling point
  - return value
    - we can also use registers for that (in x86, we use RAX)
Stack Frames

- We use two registers to manage stack and stack frames
  - RSP (stack pointer) register points to the top of the stack
  - RBP (base pointer) points to the current frame
Calling Convention

• Calling conventions define how parameters and return values are exchanged between a caller function and the called function (callee)

• In principle, you can define your own calling conventions

• However, if you want to interoperate with functions and libraries written by others, everyone needs to follow a common calling convention
C/x86-64 Calling Convention

• Caller
  – puts first six arguments into registers
  – puts remaining arguments (if any) onto the stack
  – invokes callee function by using call instruction

• Callee
  – saves key registers for caller
  – makes room for local variables
  – does work
  – puts return value into register (EAX)
  – cleans up stack frame and restores key registers
  – invokes ret(urn) call
We want to call a function
We want to call a function

- Caller might push parameters on the stack (if there are more than 6)
  - this is done in reverse order, right to left: first push last parameter
We want to call a function

- Caller might push parameters on the stack (if there are more than 6)
  - this is done in reverse order, right to left: first push last parameter

- Then, invoke function (via call)
  - when this is done, the return address is automatically pushed on the stack
C/x86-64 Calling Convention

- In the function prologue of the callee, we first save the old frame (base) pointer
  
push %rbp
C/x86-64 Calling Convention

• In the function prologue of the callee, we first save the old frame (base) pointer
  
  push %rbp

• Then, we copy the stack pointer value into EBP to get our new base pointer
  – allows access to parameters
  
  mov %rsp, %rbp
C/x86-64 Calling Convention

- Stack frame holds all local variables

- We need to make room
  - simply move the stack pointer (downwards)
  - for example, if we need space for two 64-bit integers
    - `sub $16, %rsp`
  - the `sub` is sometimes omitted when function is a leaf function
  - EBP is used as anchor to access local variables
C/x86-64 Calling Convention

When function is done

- Store return value in RAX
- Reset the stack
  
  ```asm
  mov %rbp, %rsp
  ```
When function is done

- Store return value in RAX
- Reset the stack
  \[ \text{mov} \ %\text{rbp}, \ %\text{rsp} \]
- Restore old frame pointer
  \[ \text{pop} \ %\text{rbp} \]
- Now we are ready for ret
Function Call Example

```c
int foo(int a, int b)
{
    int i = 3;
    return (a + b) * i;
}

int main()
{
    int e = 0;
    e = foo(4, 5);
    printf("%d", e);
}
```

Arguments are passed in registers (%rdi, %rsi)
(gdb) disassemble main
Dump of assembler code for function main:

0x0000555555555150 <+4>: push %rbp
0x0000555555555151 <+5>: mov %rsp,%rbp
0x0000555555555154 <+8>: sub $0x10,%rsp
0x0000555555555158 <+12>: mov $0x0,-0x4(%rbp)
0x000055555555515f <+19>: mov $0x5,%esi
0x0000555555555164 <+24>: mov $0x4,%edi
0x0000555555555169 <+29>: call 0x555555555129 <foo>
0x000055555555516e <+34>: mov %eax,-0x4(%rbp)
0x0000555555555171 <+37>: mov $0x0,%eax
0x0000555555555176 <+42>: leave
0x0000555555555177 <+43>: ret

End of assembler dump.

Arguments are passed in registers (%rdi, %rsi)
Breakpoint 1, 0x0000555555555129 in foo ()
(gdb) disassemble
Dump of assembler code for function foo:

```
0x0000555555555129 <+0>:  endbr64
0x000055555555512d <+4>:  push   %rbp
0x000055555555512e <+5>:  mov    %rsp,%rbp
0x0000555555555131 <+8>:  mov    %edi,-0x14(%rbp)
0x0000555555555134 <+11>: mov    %esi,-0x18(%rbp)
0x0000555555555137 <+14>: movl   $0x3,-0x4(%rbp)
0x000055555555513e <+21>: mov    -0x14(%rbp),%edx
0x0000555555555141 <+24>: mov    -0x18(%rbp),%eax
0x0000555555555144 <+27>: add    %edx,%eax
0x0000555555555146 <+29>: imul   -0x4(%rbp),%eax
0x000055555555514a <+33>: pop     %rbp
0x000055555555514b <+34>: ret
```

End of assembler dump.
The “foo” Frame

Breakpoint 2, 0x0000555555555137 in foo ()
(gdb) stepi
0x000055555555513e in foo ()
(gdb) x/8w $rbp-8
0x7fffffffdfa8: 0x55555180 0x00000003 0xfffffd0 0x00007fff
0x7fffffffdfb8: 0x5555516e 0x00005555 0xffffe0c0 0x00007fff

$rbp

$rbp-8
Part II
Taking Control of the Program
The Idea

- Overwrite a pointer with the address of our code
- First, locate a pointer that will be copied to the RIP register, or that points to the data that will be copied to the RIP
  - function return address
  - function pointers
  - saved RBP
  - entry in the GOT (Global Offset Table)

- Second, overwrite pointer with “good” value
  - we will see what good value means
Smashing the Stack

- A procedure contains local variable allocated on the stack
- Procedure copies user-controlled data (input) to the buffer without verifying that the data size is smaller than the buffer
- The user data overwrites all other variables on the stack, up to the return address
- Procedure returns, program fetches the return address that has been modified and jumps to it
Example

$ cat test.c
#include <stdio.h>
#include <string.h>

int vulnerable(char* param)
{
    char buffer[100];
    strcpy(buffer, param);
}

int main(int argc, char* argv[])
{
    vulnerable(argv[1]);
    printf("Everything's fine\n");
}

$ gcc -fno-stack-protector -o test test.c

Buffer that can hold up to 100 bytes
Copy an arbitrary number of characters from param to buffer
$ ./test hello
Everything's fine

$ ./test
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
What Just Happened?

GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
(gdb) r hello
Everything's fine

(gdb) r
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x000005555555518e in vulnerable ()

(gdb) disassemble
Dump of assembler code for function vulnerable:
  0x000055555555518d <+36>: leave
=> 0x000055555555518e <+37>: ret
End of assembler dump.

(gdb) x $rsp
0x7fffffffdf18: 0x41414141
Choosing Where to Jump

- Address inside a buffer that contains content controlled by the attacker
  - PRO: works for remote attacks
  - CON: the attacker needs to know the address of the buffer, the memory page containing the buffer must be executable

- Address of an environment variable
  - PRO: easy to implement, works with tiny buffers
  - CON: only for local exploits, some program clean the environment, the stack must be executable

- Address of a function inside the program
  - PRO: works for remote attacks, does not require an executable stack
  - CON: need to find the right code
Jumping into a Buffer

• The buffer that we are overflowing is usually a good place to put the code that we want to execute

• The buffer is somewhere on the stack, but in most cases the exact address is unknown
  – The address must be precise: jumping one byte before or after would typically just make the application crash
  – On the local system, it is possible to calculate the address with a debugger, but it is very unlikely to be the same address on a different machine
  – Any change to the environment variables affect the stack position
Two Steps

1. Get an estimate

```c
$ cat get_sp.c
#include <stdio.h>
void* get_sp(void)
{
    __asm__("mov %rsp, %rax");
}
int main(int argc, char **argv)
{
    printf("Stack pointer (RSP): %p
", get_sp());
    return 0;
}
$ ./get_sp
Stack pointer (RSP): 0x7ffe6e093220
```

2. Make guess robust to errors: Rather than having to hit precisely, hitting somewhat close is enough (NOP sled)

Still, trial and error is often necessary
The NOP Sled

- A sled is a “landing area” that is put in front of the shellcode.
- Must be created in a way such that wherever the program jumps into it...
  - ... it always finds a valid instruction
  - ... it always reaches the end of the sled and the beginning of the shellcode.
- The simplest sled is a sequence of no operation (NOP) instructions.
  - Single byte instruction (0x90) that does not do anything.
- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target area.
Assembling the Malicious Buffer

buf address

shellcode

90 90 90 90
90 90 90 90
90 90 90 90
90 90 90 90

ret address
base pointer
buffer
Part III
The Shellcode
Shellcode

- Sequence of machine instructions that is executed when the attack is successful

- Traditionally, the goal was to spawn a shell (that explains the name “shell code”)

- They can do practically anything
  - create a new user, change a user password
  - bind a shell to a port (remote shell)
  - open a connection to the attacker machine (reverse shell)
  ...

$ man execve

int execve(const char *pathname, char *const argv[], char *const envp[]);

DESCRIPTION
execve() executes the program referred to by pathname. This causes the program that is currently being run by the calling process to be replaced with a new program.

pathname must be either a binary executable, or a script starting with a line of the form: #!interpreter [optional-arg]

argv is an array of argument strings passed to the new program. By convention, the first of these strings (i.e., argv[0]) should contain the filename associated with the file being executed.

envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program.

The argv and envp arrays must each include a null pointer at the end of the array.
How to Spawn a Shell

$ man execve

```c
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

**DESCRIPTION**

`execve()` executes the program referred to by `pathname`. This causes the program that is currently being run by the calling process to be replaced with a new program.

`pathname` must be either a binary executable, or a script starting with a line of the form: `#!interpreter [optional -arg]`

`argv` is an array of argument strings passed to the new program. By convention, the first of these strings (i.e., `argv[0]`) should contain the filename associated with the file being executed.

`envp` is an array of strings, conventionally of the form key=value, which are passed as environment to the new program.

The `argv` and `envp` arrays must each include a null pointer at the end of the array.
How to Spawn a Shell

- Let’s call the execve system call directly
- Use the `syscall` instruction
- `execve` syscall has number 59 (integer) - 0x3b (hex)
- System call number goes into RAX
- Similar to function calls, arguments passed in registers (RDI, RSI, RDX)

- Shellcode needs to set up the arguments, load system call number, and invoke syscall instruction
How to Spawn a Shell

Shellcode needs to set up the arguments, load system call number, and finally invoke `syscall` instruction.

```
xor %rax, %rax ; set RAX to 0
push %rax ; push '\0' (will be string terminator)
mov $0x68732f6e69622f2f, %rax ; move "hs/nib/" into RAX
push %rax ; push "//bin/sh\0" on the stack
mov %rsp, %rdi ; 1st arg (RDI): RSP points to //bin/sh\0
xor %rsi, %rsi ; 2nd arg (RSI): NULL
xor %rdx, %rdx ; 3rd arg (RDX): NULL
xor %rax, %rax
mov $0x3b, %al
syscall ; 0x3b in RAX -> syscall will be execve
; invoke execve() with '/bin/sh'
```
The Zeros Problem

- The shellcode is usually copied into a string buffer
- `\x00` is the string terminator character
- Problem: any null byte in the shellcode would stop copying

- One solution: substitute any instruction containing zeros with an alternative instruction
  
<table>
<thead>
<tr>
<th>Original Instruction</th>
<th>Alternative Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov 0x0, reg</code></td>
<td><code>xor reg, reg</code></td>
</tr>
<tr>
<td><code>mov 0x1, reg</code></td>
<td><code>xor reg, reg</code></td>
</tr>
<tr>
<td></td>
<td><code>inc reg</code></td>
</tr>
</tbody>
</table>

- Alternative solution to modifying shellcode: staging
  - encode shellcode (e.g., base64, eliminate unwanted chars)
  - decode before jumping to original code
Exploit Considerations

• You might want more powerful shellcode
  – typically, you don’t write it yourself
  – there are generator tools, such as pwntools or metasploit/venom

• If you want to develop on your own machine
  – you want to disable OS and compiler defenses (see next section)
    • echo 0 > /proc/sys/kernel/randomize_va_space
    • gcc -fno-stack-protector -z execstack -o program program.c
DEFENSES AND EVOLUTION OF ATTACKS
Defense in Depth

- **Program / programmer level**
  - write safe code
  - static (source) code analysis

- **Compiler and run-time level**
  - safe libraries (that add extra checks)
  - stack protection (stack canaries)
  - control flow integrity (CFI)

- **Operating system level**
  - data execution protection (DEP)
  - address space layout randomization (ASLR)
Stack Protection

• Goal
  – protect the function frame from being overwritten by the attacker

• Idea
  – add a "canary" value between the local variables and the saved EBP
  – at the end of the function, check that the canary is “still alive”
  – a different canary value means that a buffer preceding it in memory has been overflowed
Canary Values

• Terminator canaries: contain string terminator characters (\0) to stop string copy routines

• Random canaries: contain a random value generated at program initialization and stored in a global variable
  – the attacker has to find a way to read the canary

• Random XOR canaries: contain a random value XORed with all (or part of) the control data to protect
  – can be used to detect attacks in which the attacker is able to modify the return address without overwriting the canary
Stack Protection Implementations

• **StackGuard**
  – first canary implementation (by Immunix Corp) in 1997
  – implemented as a patch for gcc 2.95

• **GCC Stack-Smashing Protector (ProPolice)**
  – first developed as a patch for gcc 3.x
  – supports canary and stack variable rearrangement
  – part of gcc 4.1

• **Visual Studio 2003 - GS option**
  – compiler option to insert canaries (called security cookies by Microsoft), stack rearrangement
Stack Protection in gcc

- **-fstack-protector**
  - StackGuard and ProPolice (more modern)
  - ProPolice – also makes sure that stack pointers are put at lower addresses than buffers (why is that smart?)

- **-fstack-protector-strong** from gcc 4.9

- **-fno-stack-protector**
  - Deactivate it, good for practicing buffer overflows
Data Execution Prevention (DEP)

• Does not block buffer overflows, but prevents the shellcode from being executed
  – ensure that data (on heap or stack cannot be executed)
  – might affect the execution of some programs that normally require to execute data on the stack (trampolines)

• Supported by most operating systems
  – originally implemented in software by PaX on Linux, closely followed by OpenBSD W^X
  – modern implementations rely on hardware support (e.g., ia32/x86_64 NX bit, tagged memory)
Code Reuse

- Idea: Instead of injecting a payload, construct an exploit by reusing existing code

- Application and library code must be executable, thus, DEP does not apply

- Desired functionality must be present in addressable memory
  - Simplest example: Return-into-libc
  - More general approach: Return-oriented programming (ROP)
Return-into-libc

• The shellcode in the buffer cannot be executed but ...  
  – the attacker can still control the stack content  
  – thus, the attacker can control the RIP value

• Why not call existing code?

• libc is an attractive target  
  – very powerful functions (system(), execve()...)  
  – linked by almost every program
Return-into-libc

UC Santa Barbara

- function param
- return address
- saved rbp
- buffer

- addr of /bin/sh
- system() addr
- ..... (system() return address)

buffer overflow

function returns
Return-Oriented Programming

• Return-oriented programming (ROP) extends return-into-libc
  – introduced by Hovav Shacham in 2007
  – shown to be Turing complete (for libc)
  – in practice, it is used to bypass memory protections

• Instead of reusing functions, ROP reuses code gadgets
  – gadgets are small sequences of instructions ending in a return
  – each gadget performs some small update to the program state
  – execution becomes a chain of returns to gadgets
Goal: eax = eax - 1

- &gadget_0
  - xor edx, edx
  - ret

- &gadget_1
  - inc edx
  - ret

- &gadget_2
  - sub eax, edx
  - ret
Goal: \*addr = 0xdeadbeef

...  
&gadget_2  
addr  
&gadget_1  
0xdeadbeef  
&gadget_0

mov [ecx], ebx  
ret  

pop ecx  
ret  

pop ebx  
ret
Gadget Extraction

89 50 04 a3 ff d0 05 08 83 c4 04 5b 5d c3
Gadget Extraction

```
mov[ eax+4], edx
mov 0x805d0ff, eax
add esp, 4
pop ebx
ret
```
Gadget Extraction

add al, 0xa3    call [eax]

mov[eax+4], edx  mov 0x805d0ff, eax  add esp, 4

pop ebp  pop ebx  ret
Gadget Extraction

```
add al, 0xa3   call [eax]
add al, 0x5b   ret
```

```
mov[eax+4], edx  mov 0x805d0ff, eax  add esp, 4
```

```
add esp, 4
pop ebx  ret
```
Return-Oriented Programming

• Works against virtually every architecture
• Useful in many situations
  – non-executable memory regions
  – signed code
• When combined with memory disclosure vulnerabilities, ROP is very difficult to defend against
• State of the art in exploit development
Address Space Layout Randomization

• Introduce artificial diversity by randomly arranging the positions of key data areas (base of the executable, position of libraries, heap, and stack)
  – prevent the attacker from being able to easily predict target addresses

• Idea: Randomize code and data addresses to make their locations difficult to predict
  – adversaries must now find the location of injected code
  – adversaries now cannot easily reuse code

• Coarse-grained ASLR ⇒ random segment base offsets
  – implemented in virtually all modern operating systems
Defeating ASLR

Coarse-grained ASLR on 64-bit architectures is a strong defense, but can still be circumvented

- If any addresses or known code or data is leaked, segment offsets can easily be recovered
- Spraying can reduce non-determinism (e.g., heap spraying)
- Fixed structures sometimes remain despite ASLR
Heap Spraying

• Overwriting a function pointer is often easily achieved

• Idea: Instead of getting the address exactly right, try to increase the chance of hitting shellcode
  – force allocation of many memory objects containing shellcode
Heap Spraying

• Process layout (32-bit Linux systems)

0x00000000 – 0xbfffffff: user memory

.code

foo() bar()

heap

buf

stack

0xc0000000 – 0xffffffff: kernel memory
Heap Spraying

- Process layout (32-bit Linux systems)

<table>
<thead>
<tr>
<th>Memory Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 - 0xbfffffff</td>
<td>User memory</td>
</tr>
<tr>
<td>0xc0000000 - 0xffffffff</td>
<td>Kernel memory</td>
</tr>
</tbody>
</table>

Diagram:

- .code
  - foo()
  - bar()

- heap
  - buf

- stack

- 86
Heap Spraying

• Process layout (32-bit Linux systems)

0x00000000 – 0xffffffff: user memory

0xc0000000 – 0xffffffff: kernel memory
Heap Spraying

- Process layout (32-bit Linux systems)

0x00000000 – 0xffffffff: user memory

0xc0000000 – 0xffffffff: kernel memory
Heap Spraying

- Process layout (32-bit Linux systems)

![Diagram showing process layout]

- User memory: 0x00000000 – 0xbfffffff
- Kernel memory: 0xc0000000 – 0xffffffff
Heap Spraying

- **Requirement**
  - we need control over memory allocations
  - must create many objects containing shellcode

- **Solution: embedded scripts**
  - today, many applications allow execution of user-provided scripts in the context of the application/document to enrich usability
  - JavaScript (browsers, PDF readers)

- **Before exploiting a memory corruption bug, allocate many objects (e.g., strings) filled with shellcode**
Control Flow Integrity (CFI)

• A control transfer is allowed $\iff$ the control transfer is present in the original program

• First stage: Determine legal control transfers
  – extract control flow model from a program
  – using static code analysis, could be part of compiler pass

• Second stage: Enforce that only legal control transfers occur at run-time
  – add runtime checks at call sites
  – program terminated if a check fails
OTHER MEMORY CORRUPTION ATTACKS
Heap Overflows

- The heap is the area of memory that is dynamically allocated through the “malloc” family functions
  - malloc(), calloc(), realloc(), free()
  - new(), delete()
  - functions that return dynamically allocated memory, e.g., strdup()
- These functions request memory from the kernel by invoking various syscalls (e.g., brk(), mmap()..)
- The heap grows towards higher memory addresses
- The allocation algorithm is OS/version-dependent
Same General Idea as with the Stack

- Memory management is done through in-band control structures (metadata) also stored on the heap
  - Usually contains data like pointers, size values, indexes into arrays, ...
  - It’s usually stored right before the piece of data that has been requested

- When two (or more) free pieces of memory are next to each other, they are merged into one bigger piece of free memory (to avoid fragmentation)
Heap Overflow Vulnerabilities

• First demonstrated by Solar Designer on 25 July 2000
  – JPEG COM Marker Processing Vulnerability in Netscape Browsers

• General way to exploit heap overflow in order to execute arbitrary code on the machine
  – Main idea is to attack the memory management algorithm, taking advantage of the mixing of data and control information on the heap

• Evolved into a key vulnerability in systems software
  – Microsoft reported that 53% of their security problems in 2017 were heap-related vulnerabilities
Integer Overflows

• Integer overflows are caused by unexpected results when comparing, casting, and adding integers

• Integer overflow and underflow
  – The result of an arithmetic operation lies outside the range of the variable type
  – Example:
    ```c
    short x = 0x7FFF; x++; /* x is now -32768 */
    ```

• Casting errors
  – Casting signed to/from unsigned
  – Casting to type of different size
  – Example:
    ```c
    unsigned long l; short x = -2; l = x;
    /* l is now 4294967294 */
    ```
int main(int argc, char *argv[]) {
    char buf[512];
    long max;
    short len;
    max = sizeof(buf);
    len = strlen(argv[1]);
    printf("max %ld len %d\n", max, len);
    if (len < max) {
        strcpy(buf, argv[1]);
    }
}
$ ./integeroverflow `python -c 'print "A" * 32000'`
max 512 len 32000

$ ./integeroverflow `python -c 'print "A" * 33000'`
max 512 len -32536
Segmentation fault (core dumped)
Format String Vulnerabilities

int printf(const char *format, ...)  

• The first parameter (format) is the format string  
  – It can contain normal text (copied in the output)  
  – It can contain placeholders for variables  
    • Identified by the character %  
    • The corresponding variables are passed as arguments  

• Example:  
  • printf(“X = %d”, x);
The printf Function

• Different placeholders for different variable types
  – %s string
  – %d decimal number
  – %f float number
  – %c character
  – %x number in hexadecimal form
  – ..... 

• If the attacker can control the format string, she can overwrite any location in memory

• All the members of the family are vulnerable: fprintf, sprintf, vfprintf, vprintf, vsnprintf...
A Vulnerable Program

int main(int argc, char* argv[]) {
    char buf[256];
    snprintf(buf, 250, argv[1]);
    printf("buffer: %s\n", buf);
    return 0;
}
A Vulnerable Program

```c
int main(int argc, char* argv[]) {
    char buf[256];

    snprintf(buf, 250, argv[1]);
    printf("buffer: %s\n", buf);
    return 0;
}
```

```
> ./format hello
buffer: hello

> ./format "hello |%x %x %x|
buffer: hello |affff874 a7ff2d29 a7eb3aab|
```
• \%n: writes the number of bytes printed so far in the address specified as parameter

> ./format "AAAA \%x \%x \%x \%x \%x \%x \%x \%n"

buffer: AAAA affff864 a7ff2d29 a7eb3aab 80482180 0 0 8048184 41414141

./format "AAAA \%x \%x \%x \%x \%x \%x \%x \%n"

\%n gets an address from the stack (in the example 0x41414141) and writes the number of characters printed so far to it, as if it was a pointer to an integer variable