CS 177 - Computer Security

Memory Corruption
Agenda

• Intel x86 (32-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 (32-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. mobile phones)
  - This architecture is fundamental, has not changed

- What is memory used for?
  - E.g., location of cursor, 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 32-bit x86 architecture
• Illustrates principles well
  – similar problems appear on other architectures
• Simpler to exploit
Intel x86 Architecture

- Very popular, but “crazy”
- CISC (complex instruction set computing)
  - over 100 distinct opcodes in the set
- Register-poor
  - only six general purpose 32-bit registers
- Variable-length instructions
- Built of many backwards-compatible revisions
x86 Registers

• General purpose registers:
  – %eax, %ebx, %ecx, %edx, %edi, %esi

• Special purpose registers:
  – %ebp, %esp, %eip, %eflags

• Some registers (e.g., %eip, %eflags) can only be accessed through special instructions
Program Counter (%eip)

• 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)

• Instruction ends with data length (l = long = 32 bits)
• Format is: opcode, src, dst

• Constants preceded by $
  movl $16, %ebx

• Registers preceded by %
  movl %eax, %ebx

• Indirection uses ( )
  movl (%eax), %ebx
  movl 4(%eax), %ebx

  The item stored at %eax + 4
Process Memory Layout

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

**Example:** Outside of any function:

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

**Example:** Outside of any function:

```c
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, 32-bit) address space.
sub Instruction

- Subtract from a register value

```
subl %eax, %ebx
```

Registers:
- %eax: 7
- %ebx: 9

Memory:
```
subl %eax, %ebx
```
push Instruction

- Put a value on the stack
  - value from register
  - value goes to (%esp)
  - subtract 4 from %esp
- Example
  ```
pushl %eax
  ```
pop Instruction

- Take value from the stack
  - value from (%esp)
  - value goes into register
  - add 4 to %esp

- Example
  popl %eax

\[ \begin{align*}
  &\text{registers} \\
  &\begin{array}{ccc}
    \%eax & 9 \\
    \%ebp & M \\
    \%esp & N
  \end{array} \\
  &\text{memory} \\
  &\begin{array}{c}
    \text{Stack}
  \end{array} \\
\end{align*} \]
• Control flow transfer
  – %eip points to the currently executing instruction (in the text section)
  – Has unconditional and conditional forms
  – Example uses relative addressing
**call Instruction**

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

### Diagram

- **%eip (K)**:
  - Points to the next instruction
  - Initial value before call

- **%ebp (M)**:
  - Points to the stack frame base
  - Value before call

- **%esp (N)**:
  - Points to the stack pointer
  - Value before call

### Registers

- **%eip**: Points to the next instruction after call
- **%ebp**: Points to the stack frame base after call
- **%esp**: Points to the stack pointer after call

### Memory

- **foo**: Points to the first instruction of the called function
- **A+i**: Points to the argument value
- **Stack**: Shows the stack before and after the call
ret Instruction

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

```
%eip  K  %ebp  M  %esp  N
```

```
K: ret
```

```
%eip  A  %ebp  M  %esp  N+4
```

```
A: instr.
```

```
Stack
```
leave Instruction

- Prepare return
- Equivalent to
  - `movl %ebp, %esp`
  - `popl %ebp`
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
  - function parameters
    - items passed to function for processing
    - could also use registers (in i64, this happens for some arguments)
  - 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 %eax)
Stack Frames

• We use two registers to manage stack and stack frames
  – %esp (stack pointer) register points to the top of the stack
  – %ebp (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 Calling Convention

• Use stack to pass parameters and to manage stack frames
  – caller
    • puts arguments on the stack
    • invokes callee by using call instruction
  – callee
    • saves key registers for caller (including old %ebp)
    • 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
C/x86 Calling Convention

We want to call a function

%ebp
%esp

Previous frame

Free memory
We want to call a function

- Caller pushes parameters on the stack
  - this is done in reverse order, right to left: first push n\textsuperscript{th} parameter, ..., second, and finally, first parameter
We want to call a function

• Caller pushes parameters on the stack
  – this is done in reverse order, right to left: first push $n^{th}$ parameter, ..., second, and finally, first parameter

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

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

- In the function prologue of the callee, we first save the old frame (base) pointer
  
  \[
  \text{push } \%\text{ebp}
  \]

- Then, we copy the stack pointer value into \%ebp to get our new base pointer
  - allows access to parameters
  
  \[
  \text{mov } \%\text{esp}, \%\text{ebp}
  \]

<table>
<thead>
<tr>
<th>Previous frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter #3</td>
</tr>
<tr>
<td>Parameter #2</td>
</tr>
<tr>
<td>Parameter #1</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>Old frame pointer</td>
</tr>
</tbody>
</table>

%ebp %esp

Free memory
C/x86 Calling Convention

- Stack frame holds all local variables

- We need to make room
  - simply move the stack pointer (downwards)
  - for example, if we need two (32-bit) words
    * subl $8, %esp
  - %ebp is also used as anchor to access local variables

```
%ebp
%esp
```

```
Previous frame
 Parameter #3
 Parameter #2
 Parameter #1
 Return address
 Old frame pointer
 Local variable #1
 Local variable #2
 Free memory
```
C/x86 Calling Convention

We function is done

- Store return value in %eax
- Reset the stack
  
  \texttt{movl \%ebp, \%esp}

![Diagram of C/x86 calling convention](image)
We function is done

- Store return value in %eax
- Reset the stack
  ```assembly
  movl %ebp, %esp
  ```
- Restore old frame pointer
  ```assembly
  popl %ebp
  ```
- 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);
}
```
A Closer Look

(gdb) disas main
Dump of assembler code for function main:
0x0804836d <main+0>:   push   %ebp
0x0804836e <main+1>:   mov    %esp,%ebp
0x08048370 <main+3>:   sub    $0x18,%esp
0x08048373 <main+6>:   and    $0xfffffffff0,%esp
0x08048376 <main+9>:   mov    $0x0,%eax
0x0804837b <main+14>:  add    $0xf,%eax
0x0804837e <main+17>:  add    $0xf,%eax
0x08048381 <main+20>:  shr    $0x4,%eax
0x08048384 <main+23>:  shl    $0x4,%eax
0x08048387 <main+26>:  sub    %eax,%esp
0x08048389 <main+28>:  movl   $0x0,0xfffffffc(%ebp)
0x08048390 <main+35>:  movl   $0x5,0x4(%esp)
0x08048398 <main+43>:  movl   $0x4,(%esp)
0x0804839f <main+50>:  call   0x8048354 <foo>
0x080483a4 <main+55>:  mov    %eax,0xfffffffffc(%ebp)
(gdb) breakpoint foo
Breakpoint 1 at 0x804835a
(gdb) run
Starting program: ./test1
Breakpoint 1, 0x0804835a in foo ()
(gdb) disas
Dump of assembler code for function foo:
0x08048354 <foo+0>:    push   %ebp
0x08048355 <foo+1>:    mov    %esp,%ebp
0x08048357 <foo+3>:    sub    $0x10,%esp
0x0804835a <foo+6>:    movl   $0x3,0xfffffffc(%ebp)
0x08048361 <foo+9>:    mov    0xc(%ebp),%eax
0x08048364 <foo+12>:   add    0x8(%ebp),%eax
0x08048367 <foo+15>:   imul   0xfffffffff(%ebp),%eax
0x0804836b <foo+19>:   leave
0x0804836c <foo+20>:   ret
End of assembler dump.
(gdb)
The “foo” Frame

(gdb) stept
0x08048361 in foo ()
(gdb) x/12wx $ebp-16
0xaf9d3cc8: 0xaf9d3cd8 0x080482de 0xa7faf360 0x00000003
0xaf9d3cd8: 0xafdde9f8 0x080483a4 0x00000004 0x00000005
0xaf9d3ce8: 0xaf9d3d08 0x080483df 0xa7fadff4 0x08048430

5
4
0x080483a4
0xafdde9f8
3
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 EIP register, or that points to the data that will be copied to the EIP
  – function return address
  – function pointers
  – saved EBP
  – 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
/** Test2.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");
}
Let's Make it Crash

> ./test2 hello
  Everything's fine

> ./test2
  Everything's fine

> ./test2
  Everything's fine

> ./test2
  Everything's fine

> ./test2
  Everything's fine

Segmentation fault

>
> gdb ./test2

(gdb) run hello
Starting program: ./test2
Everything's fine

(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!...
Starting program: ./test2 AAAAAAAAAA...
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
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 a 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
Jumping into a Buffer

**Two Steps**

1. Get an estimate

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

UC Santa Barbara

[Diagram showing buffer, base pointer, params, ret address, shellcode, and buf address with hexadecimal values 90 90 90 90 repeated]

90 90 90 90
90 90 90 90
90 90 90 90
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)
  …
How to Spawn a Shell

```c
void main(int argc, char **argv) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

(gdb) disas execve
....
mov 0x8(%ebp),%ebx
mov 0xc(%ebp),%ecx
mov 0x10(%ebp),%edx
mov $0xb,%eax
int $0x80
....
How to Spawn a Shell

int execve(char *file, char *argv[], char *env[])

(gdb) disas execve
....
mov 0x8(%ebp),%ebx
mov 0xc(%ebp),%ecx
mov 0x10(%ebp),%edx
mov $0xb,%eax
int $0x80
....

copy *file to ebx

invoke the syscall

copy *argv[] to ecx

put the syscall number into eax (execve = 0xb)

copy *env[] to edx
How to Spawn a Shell

• Three parameters
  – *file: put somewhere in memory the string `\bin\sh` (terminated by `\0`)
  – *argv[]: put somewhere in memory the address of the string `\bin\sh` followed by NULL (0x00000000)
  – *env[]: put somewhere in memory a NULL
The Address Problem

• How can we put in memory the address of the string `\bin\sh` if we do not even know where the position of the shellcode is?

• Solution...
  – the `call` instruction puts the return address on the stack
  – if we put a `call` instruction just before the string `\bin\sh`, when it is executed, it will push the address of the string onto the stack
popl gets the return address set by the call instruction from the stack (that is, the address of /bin/sh)
### The Shellcode (Almost Ready)

```
jmp 0x26 # 2 bytes
popl %esi # 1 byte
movl %esi,0x8(%esi) # 3 bytes
movb $0x0,0x7(%esi) # 4 bytes
movl $0x0,0xc(%esi) # 7 bytes
movl $0xb,%eax # 5 bytes
movl %esi,%ebx # 2 bytes
leal 0x8(%esi),%ecx # 3 bytes
leal 0xc(%esi),%edx # 3 bytes
int $0x80 # 2 bytes
movl $0x1,%eax # 5 bytes
movl $0x0,%ebx # 5 bytes
int $0x80 # 2 bytes
call -0x2b # 5 bytes
.string "/bin/sh" # 8 bytes
```

**Diagram:**
- **setup**
  - `esi + 0xc`: 0x0
  - `esi + 0x8`: `esi`
- **execve()**
  - `esi`: `/bin/sh\0`
- **exit()**
- **setup**
The Zeros Problem

- The shellcode is usually copied into a string buffer
- \x00 is the string terminator character
- Problem: any null byte would stop copying
- One solution: substitute any instruction containing zeros, with an alternative instruction

\[
\begin{align*}
\text{mov 0x0, reg} & \rightarrow \text{xor reg, reg} \\
\text{mov 0x1, reg} & \rightarrow \text{xor reg, reg} \\
& \quad \text{inc reg}
\end{align*}
\]
The Zeros Problem

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

```c
char shellcode_with_NULLs[] = "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00...";
char shellcode[] = "DECODE(BASE64SHELLCODE);BASE64SHELLCODE";
```
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
  - compile target as 32-bit binaries
    - gcc -m32 -o server server.c
  - 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 server server.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

| local variables | canary | saved ebp | ret addr | params |
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 on 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 EIP value

- Why not call existing code?

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

buffer overflow

function returns

function param

ret address

saved ebp

buffer

addr of /bin/sh

.....

system() addr

.....

addr of /bin/sh

.....

system() return address
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
Gadgets

Goal: eax = eax - 1

&gadget_2
sub eax, edx
ret

&gadget_1
inc edx
ret

&gadget_0
xor edx, edx
ret
Goal: \(*\text{addr} = 0x\text{deadbeef}\)
Gadget Extraction

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

pop ebp  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]

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

add al, 0x5b    pop ebp    ret

pop ebp    ret

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
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 – 0xffffffff: user memory

.code
foo()

bar()

heap
buf

stack

0xc0000000 – 0xffffffff: kernel memory
```
Heap Spraying

- Process layout (32-bit Linux systems)

```
0x00000000 – 0xffffffff: user memory
  .code  foo()  bar()

heap  buf

stack

0xc0000000 – 0xffffffff: kernel memory
```
Heap Spraying

- Process layout (32-bit Linux systems)

```plaintext
0x00000000 – 0xbfffffff: user memory

.code

foo()

bar()

heap

buf

stack

0xc0000000 – 0xffffffff: kernel memory
```
Heap Spraying

- Process layout (32-bit Linux systems)

```
0x00000000 – 0xbfffffff: user memory

.code
  foo()
  bar()

heap
  buf
  buf
  buf
  buf
  buf
  buf

stack

0xc0000000 – 0xffffffff: kernel memory
```
Heap Spraying

- Process layout (32-bit Linux systems)

```
0x00000000 – 0xbfffffff: user memory
  .code  foo()  bar()

heap
  buf  buf  buf  buf  buf  buf

stack

0xc0000000 – 0xffffffff: kernel memory
```
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)
  – ActionScript (flash applications)

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

• A control transfer is allowed ⇐⇒ 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:
    
    ```
    short x = 0x7FFF; x++; /* x is now -32768 */
    ```
- Casting errors
  - Casting signed to/from unsigned
  - Casting two type of different size
  - Example:
    
    ```
    unsigned long l; short x = -2; l = x;
    /* l is now 4294967294 */
    ```
Integer Overflows

```c
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]);
    }
}
```
Integer Overflows

$ ./integeroverflow `python -c 'print "A" * 32000'``
max 512 len 32000
$ ./integeroverflow `python -c 'print "A" * 33000'``
max 512 len -32536
Segmentation fault
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

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

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 %x"
buffer: AAAA affff864 a7ff2d29 a7eb3aab 8048218 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