Week 6: Lecture A
Defending Applications

Tuesday, September 26, 2023
Announcements

- **Project 2: AppSec** released
  - **Deadline:** Thursday, October 19th by 11:59PM
Project 2 progress?

- Finished with Targets 0-1!
- Working on Targets 0-1...
- Haven't started :(
Announcements

- **Project 1** grades are now available on Canvas

- **Statistics:**
  - Average score: **85%**

- **Fantastic job!**
Announcements

- **Project 1** grades are now available on **Canvas**

- Think we made an error? Request a regrade!
  - Valid regrade requests:
    - You have verified your solution is correct
      (i.e., we made an error in grading)

**Project 1 Regrade Requests** (see **Piazza** pinned link):
Submit by **11:59 PM** on **Monday 10/02** via **Google Form**
Announcements

See Discord for meeting info!

www.utahsec.com
Questions?
Last time on CS 4440...

Shellcode
Constructing Exploits
Pointer Dereferences
Integer Overflows
**Attacker goal:** make program open a **root shell**
- Root-level permissions = **total system ownage**
- **You’ll do this in Project 2!**

**Shellcode** = code to open a root shell
- Inject this somewhere and **direct execution to it**
- Basic structure:
  1. Call `setuid(0)` to set user ID to “root”
  2. Open a shell with `execve("/bin/sh")`

```c
setuid(0) + execve("/bin/sh")
```
Exploiting Buffer Overflows

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**

<table>
<thead>
<tr>
<th>foo()'s return addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()'s frame ptr</td>
</tr>
<tr>
<td>char * buffer[16]</td>
</tr>
</tbody>
</table>
**Exploiting Buffer Overflows**

- **Key idea:** inject evil code inside buffer, and *redirect execution to it*

<table>
<thead>
<tr>
<th>foo()'s return addr</th>
<th>main()'s frame ptr</th>
<th>char * buffer[16]</th>
</tr>
</thead>
</table>

Evil code here!
Exploiting Buffer Overflows

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**

```
| foo()'s return addr |
| main()'s frame ptr |
| char * buffer[16]  |
```

Padding to reach RetAddr

Evil code here!
### Exploiting Buffer Overflows

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**

<table>
<thead>
<tr>
<th>foo()'s return addr</th>
<th>Start addr of buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()'s frame ptr</td>
<td>Padding to reach RetAddr</td>
</tr>
<tr>
<td>char * buffer[16]</td>
<td>Evil code here!</td>
</tr>
</tbody>
</table>
**Exploiting Buffer Overflows**

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**

```plaintext
<table>
<thead>
<tr>
<th>foo()'s return addr</th>
<th>Start addr of buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()'s frame ptr</td>
<td>Padding to reach RetAddr</td>
</tr>
<tr>
<td>char * buffer[16]</td>
<td>Evil code here!</td>
</tr>
</tbody>
</table>
```

When `foo()` returns, execution will proceed to our **buffer’s address**...

Thus **executing our evil code**!
Bounded vs. Unbounded Writes

- **Targets 0–2** permit **unbounded** writes
  - We can overwrite **anything** in the higher stack memory
  - Thanks to dangerous functions `gets()` and `strcpy()`
  - Definitely don’t use these functions in your own code!

- **Targets 3–4** are **bounded** writes... limited reach!
  - **Target 3**: we can only write `8 + sizeof(buf)` bytes
  - **Target 4**: we can only write `count` bytes (via `fread()`)

Stefan Nagy
Bounded vs. Unbounded Writes

- **Targets 0–2** permit **unbounded** writes
  - We can overwrite anything in the higher stack memory
  - Thanks to dangerous functions `gets()` and `strcpy()`
  - Definitely don’t use these functions in your own code!

- **Targets 3–4** are **bounded** writes... limited reach!
  - **Target 3**: we can only write $8 + \text{sizeof}(\text{buf})$ bytes
  - **Target 4**: we can only write `count` bytes (via `fread()`)

For **bounded** writes, we have to get creative and **find a way to overwrite** what we want!
Memory Addresses Point to Memory Slots

Key idea: it’s all “things” pointed to by addresses

Example: instructions in the Program Text:

```assembly
$ disas vulnerable:
0x0804a17b <+0>: endbr32
0x0804a17f <+4>: push %ebp
0x0804a180 <+5>: mov %esp,%ebp
0x0804a182 <+7>: push %ebx
```
Memory Addresses Point to Memory Slots

Key idea: it’s all “things” pointed to by addresses

<table>
<thead>
<tr>
<th>Memory Region</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Virtual Memory</td>
<td></td>
</tr>
<tr>
<td>Stack Memory</td>
<td>unmapped</td>
</tr>
<tr>
<td>Shared Libraries</td>
<td>unmapped</td>
</tr>
<tr>
<td>Heap Memory</td>
<td></td>
</tr>
<tr>
<td>Uninitialized Data</td>
<td></td>
</tr>
<tr>
<td>Initialized Data</td>
<td></td>
</tr>
<tr>
<td>Program Text</td>
<td></td>
</tr>
</tbody>
</table>

Example: payload NOPs in **Stack Memory**:

```
$ x/32xw 0xffff6d8cc
0xffff6d8cc : 0x90909090 0x90909090
0xffff6d8d4 : 0x90909090 0x90909090
0xffff6d8dc : 0x90909090 0x90909090
0xffff6d8e4 : 0x90909090 0x90909090
```
void foo(char *str) {
    int *p;
    int a;
    *p = a;
}

Contents of 0x00000000 updated to a

foo()'s retAddr

caller's EBP

Address 0x00000000

int a

Stack Addresses

EBP+4
EBP+0
EBP-4
EBP-8
0x00000000

Indirect Memory Overwrite

0x000000
void foo(char *str) {
    int *p;
    int a;
    *p = a;
}

Stack Addresses

- foo()'s retAddr: EBP+4
- caller’s EBP: EBP+0
- Address EBP+4
- Shellcode Address: EBP-4
- EBP-8
void foo(char *str) {
    int *p;
    int a;
    *p = a;
}

Contents of EBP+4 updated to the shellcode address!
- Integer overflows behave differently from stack buffer overflows
  - Really just integer “wrap-arounds”

### 32-bit Integer Range:

**Unsigned:**

\[
[0, (2^{32} - 1)]
\[
[0, 4294967295]
\]

**Signed:**

\[
[-2^{31}, (2^{31} - 1)]
\]
\[
[-2147483648, 2147483647]
\]
Integer Overflows

- **Integer overflows** behave differently from stack buffer overflows
  - Really just integer “wrap-arounds”

  
  
<table>
<thead>
<tr>
<th>32-bit Integer Range:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Unsigned: [0, (2^32 - 1)]</td>
</tr>
<tr>
<td>[0, 4294967295]</td>
</tr>
<tr>
<td>-Signed: [-2^31, (2^31 - 1)]</td>
</tr>
<tr>
<td>[-2147483648, 2147483647]</td>
</tr>
</tbody>
</table>

- Overflowing an **unsigned integer** “wraps around” to a very small integer!
  - E.g., 0xFFFFFFFF + 2 = 0x00000002
Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program’s assumptions?

- **Target 4: a potential mismatch** of buffer’s size versus the data written to it

  - If we perform an integer overflow on `count`, `alloca()` creates an artificially small buffer
  - The resulting fill operation will exceed the buffer’s size, resulting in a buffer overflow!

```c
alloca(<MAX_UINT); // allocate our buffer
fread( &buf[i], 4, count, f ); // fill buffer
```

Range of `count`:

- `[0, ¼(MAX_UINT))`
- `[0, MAX_UINT]`
Estimating the Stack

- **Identify your target function**
  - E.g., `vulnerable()` in this case

- **Each frame contains a few key things:**
  1. The function’s **return address**
     - Address of next instruction to when the current function returns
  2. The caller’s **saved frame pointer**
     - Where EBP will get “reset” to when the current function returns
  3. The function’s **local variables**
     - E.g., char `buf[100]`
     - **Find these from the source code!**

```c
void vulnerable(char *arg){
    char buf[100];
    strcpy(buf, arg);
}
```
Padding Heuristics

- **How large** is our vulnerable buffer?
  - E.g., char `buf[100]`
  - Need **at least 100 bytes** to overflow!
    - Compilers may add a few "extra" bytes for memory alignment

- **Saved EBP** = an extra **four bytes**

- **Other things above our buffer?**
  - Other locals (e.g., `count` in Target 3)
  - Passed-by-reference function args
  - Other compiler-added artifacts

<table>
<thead>
<tr>
<th>RetAddr</th>
<th>Saved EBP</th>
<th>other stuff ???</th>
<th>buf [100]</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>TBD bytes</td>
<td>~100 bytes</td>
<td></td>
</tr>
</tbody>
</table>
Write an Initial Payload

- **Use** guesstimated payload bytes as **lower bound** for an initial attempt
  - E.g., we know our payload is **104+ bytes**

- **Goal:** overwrite the return address with a **controlled, friendly payload**
  - E.g., **104 bytes** of NOP instructions

- **Did it overwrite the return address?**
  - If **yes**—SEGFAULT on **0x90909090**
  - If **not**—program terminates gracefully
## Refine your Payload

- **Keep a table** of attempts and results

1. `b'\x90' * 104` → normal exit
   - **Too little!** Didn’t overwrite anything

2. `b'\x90' * 120` → SEGV on 0x90909090
   - **Too much!** Complete RetAddr overwrite

3. `b'\x90' * 114` → SEGV on 0x8049090
   - **We’re close**—just two bytes over!
   - Our payload should be **112 bytes**

Tweak it to figure out the **exact payload size**

<table>
<thead>
<tr>
<th>4 bytes</th>
<th>TBD bytes</th>
<th>~100 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>90909090</td>
<td></td>
<td></td>
</tr>
<tr>
<td>909090909090</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90909090909090</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After finding the distance to the return address, we now must **overwrite it**
- **Recall:** the return address is our golden ticket to **controlling the program's execution**
- Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

**Approach:** pick a **known, friendly payload** and locate it in memory
- Goal is to find **the start of your buffer!**

**Helpful GDB commands:**
- `info proc mapping`
  - Locate the stack's **boundaries**
  - E.g., `0xffff6d000` to `0xfffffe000`

```bash
$ info proc mapping // list all memory segments
Start Addr    End Addr       Size    Offset    objfile
0x8048000  0x8049000  0x1000  0x0   target2
0x8049000  0x80b8000  0x6f000  0x1000    target2
0x80b8000  0x80e8000  0x30000  0x70000   target2
0x80e8000  0x80ea000  0x2000   0x9f000    target2
0x80ea000  0x80ec000  0x2000   0xa1000    target2
0x80ec000  0x810e000  0x22000   0x0   [heap]
0xfff6d000 0xffffe000  0x91000   0x0   [stack]
```
After finding the distance to the return address, we now must overwrite it
- **Recall:** the return address is our golden ticket to **controlling the program’s execution**
- Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

**Approach:** pick a **known, friendly payload** and locate it in memory
- Goal is to find the **start of your buffer!**

**Helpful GDB commands:**
- **find minAddr,maxAddr,"string"**
  - Search memory for address of **string**
  - Use **stack boundaries** from before

```bash
$ b *vulnerable+45 // breakpoint after buf filled
Breakpoint 1, 0x0804a1a8 in vulnerable... target2.c:8

$ r "AAAA" // run program with "AAAA" as its input
Breakpoint 1, 0x0804a1a8 in vulnerable... target2.c:8

$ find 0xfff6d000,0xffffe000,"AAAA"
0xfff6d8cc // this is likely where buffer begins!
0xfffed930 // when in doubt, pick the lower address
```
After finding the distance to the return address, we now must **overwrite it**.
- **Recall:** the return address is our golden ticket to **controlling the program's execution**.
- Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**.

**Approach:** pick a **known, friendly payload** and locate it in memory.
- Goal is to find the **start of your buffer**.

**Helpful GDB commands:**
- `x/32xw,0xDEADBEEF`
  - Show bytes at address `0xDEADBEEF`
  - **Inspect candidates** from previous step.

```bash
$ b *vulnerable+45 // breakpoint after buf filled
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ r "AAAA" // run program with "AAAA" as its input
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ x/32xw 0xfff6d8cc // look for "AAAA" bytes here
0xfff6d8cc: 0x41414141 0x00000000 0x00000000 ... 
0xfff6d8d0: 0x00000000 0x00000000 0x00000000 ...
```
We’re almost there!

- **By this point**, we’ve identified our **padding length** and **buffer start address**
  - Now, introduce our **shellcode** and finalize the attack payload!
We’re almost there!

- **By this point**, we’ve identified our **padding length** and **buffer start address**.
  - Now, introduce our **shellcode** and finalize the attack payload!
Other Exploitation Techniques

- **Not just return addresses!**
  - Function pointers
  - Arbitrary data
  - C++ exceptions
  - C++ objects
  - Heap memory freelist
  - Any code pointer!
Quiz Question Recap

0x0804a014 <+00>: push %ebp
0x0804a015 <+01>: mov %esp, %ebp
0x0804a017 <+03>: sub $4, %esp
0x0804a01a <+06>: mov 16(%ebp), %eax
Quiz Question Recap

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIP</td>
<td>0x0804a014</td>
</tr>
<tr>
<td>EBP</td>
<td>0xbffff440</td>
</tr>
<tr>
<td>ESP</td>
<td>0xbffff400</td>
</tr>
</tbody>
</table>

Stack Diagram

- Return Address: 0xbfffff400
- 0xbffff3fc
- 0xbffff3f8

Code Snippet:

```
0x0804a014 <+00>: push %ebp
0x0804a015 <+01>: mov %esp, %ebp
0x0804a017 <+03>: sub $4, %esp
0x0804a01a <+06>: mov 16(%ebp), %eax
```
Quiz Question Recap

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIP</td>
<td>0x0804a014</td>
</tr>
<tr>
<td>EBP</td>
<td>0xbffff440</td>
</tr>
<tr>
<td>ESP</td>
<td>0xbffff3fc</td>
</tr>
</tbody>
</table>

Stack Diagram

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0804a014</td>
</tr>
<tr>
<td>0x0804a015</td>
</tr>
<tr>
<td>0x0804a017</td>
</tr>
<tr>
<td>0x0804a01a</td>
</tr>
</tbody>
</table>

Return Address

Saved EBP

SP
**Quiz Question Recap**

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIP</td>
<td>0x0804a014</td>
</tr>
<tr>
<td>EBP</td>
<td>0xbffff3fc</td>
</tr>
<tr>
<td>ESP</td>
<td>0xbffff3fc</td>
</tr>
</tbody>
</table>

Stack Diagram

- **Return Address**: 0xbffff400
- **Saved EBP**: 0xbffff3fc
- **BP**: 0xbffff3f8
- **EIP**: 0x0804a014
Quiz Question Recap

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIP</td>
<td>0x0804a014</td>
</tr>
<tr>
<td>EBP</td>
<td>0xbffff3fc</td>
</tr>
<tr>
<td>ESP</td>
<td>0xbffff3f8</td>
</tr>
</tbody>
</table>

Stack Diagram

- Return Address: 0xbffff400
- Saved EBP: 0xbffff3fc
- 4 bytes space: 0xbffff3f8

Assembly Code:

```
0x0804a014 <+00>:   push   %ebp
0x0804a015 <+01>:   mov    %esp, %ebp
0x0804a017 <+03>:   sub    $4, %esp
0x0804a01a <+06>:   mov    16(%ebp), %eax
```
Quiz Question Recap

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIP</td>
<td>0x0804a014</td>
</tr>
<tr>
<td>EBP</td>
<td>0xbffff3fc</td>
</tr>
<tr>
<td>ESP</td>
<td>0xbffff3f8</td>
</tr>
</tbody>
</table>

Stack Diagram

- Return Address: 0xbffff400
- Saved EBP: 0xbffff3fc
- BP: 0xbffff3f8
- 4 bytes space: 0xbffff3f8
Questions?
This time on CS 4440...

Advanced Exploitation Techniques
ASLR, DEP, and Workarounds
Other Application-level Defenses
Recap: Spawning Shells

- **Attacker goal:** make program open a **root shell**
  - Root-level permissions = **total system ownage**
  - You’ll do this in **Project 2!**

- **Shellcode** = code to open a root shell
  - Inject this somewhere and **direct execution to it**
  - Basic structure:
    1. Call `setuid(0)` to set user ID to “root”
    2. Open a shell with `execve("/bin/sh")`
```c
#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```
#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
Shell Spawning in C

#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
Shell Spawning in C

#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}

Shell inherits same privileges as the original “parent” process.

If the original process run as root, shell gives access.
Shell Spawning in C

```c
#include <stdio.h>

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

Shell inherits same **privileges** as the original “parent” process.

If the original process **run as root**, shell gives **root** access.
Shell Spawning in C

```c
#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```
main:
    pushl  %ebp
    movl  %esp, %ebp
    pushl  $0
    pushl  $0
    pushl  $.LC0
    call   execve
    leave
    ret
Shell Spawning in x86 Assembly

main:
    pushl %ebp
    movl %esp, %ebp
    pushl $0
    pushl $0
    pushl $.LC0
    call execve
    leave
    ret

Like before, we want to call `execve("/bin/sh")`
Shell Spawning in x86 Assembly

Like before, we want to call `execve("/bin/sh")`

Q: How does the stack need to look for this call to work?

```
main:
pushl %ebp
movl %esp, %ebp
pushl $0
pushl $0
pushl $.LC0
call execve
leave
ret
```
Invoking a Shell

main:
  pushl  %ebp
  movl   %esp, %ebp
  pushl  $0
  pushl  $0
  pushl  $.LC0
  call   execve
  leave
  ret

main()’s locals

??????????????????????????

??????????????????????????

??????????????????????????
Invoking a Shell

main:
pushl %ebp
movl %esp, %ebp
pushl $0
pushl $0
pushl $.LC0
call execve
leave
ret

main()’s locals

d3xcv()’s 3rd arg

?????????????????????

?????????????????????
main:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   $0
    pushl   $0
    pushl   $.LC0
    call    execve
    leave
    ret

main()’s locals

arg3 = NULL

??????????????????????

???????????????????????
Invoking a Shell

main:
pushl %ebp
movl %esp, %ebp
pushl $0
pushl $0
pushl $.LC0
call execve
leave
ret

main()'s locals

arg3 = NULL
execve()'s 2nd arg
???????????????????????
main:
pushl %ebp
movl %esp, %ebp
pushl $0
pushl $0
pushl $.LC0
call execve
leave
ret

main()'s locals

| arg3   | = NULL |
| arg2   | = NULL |
| ?????? | //???  |
Invoking a Shell

main:
  pushl %ebp
  movl %esp, %ebp
  pushl $0
  pushl $0
  pushl $.LC0
  call execve
  leave
  ret

main()'s locals

- arg3 = NULL
- arg2 = NULL

execve()'s 1st arg
main:
  pushl %ebp
  movl %esp, %ebp
  pushl $0
  pushl $0
  pushl $.LC0
  call execve
  leave
  ret

.LC0:
  .string "/bin/sh"

main()’s locals

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg3</td>
<td>NULL</td>
</tr>
<tr>
<td>arg2</td>
<td>NULL</td>
</tr>
<tr>
<td>addr to</td>
<td>&quot;/bin/sh&quot;</td>
</tr>
</tbody>
</table>
Invoking a Shell

```
main:
pushl %ebp
movl %esp, %ebp
pushl $0
pushl $0
pushl .LC0
call execve
leave
ret

.LC0:
.string "/bin/sh"
```

**main()’s locals**
- arg3 = NULL
- arg2 = NULL
- addr to "/bin/sh"

**execve()’s ret addr**
How can we **prevent** **code injection** attacks?
Application Defense: Address Space Layout Randomization
Caveats

- Our provided shellcode requires an **executable buffer**

### Diagram

- **Start addr of buffer**
- **Padding to reach RetAddr**
  - `NOP, NOP, NOP, NOP, NOP, NOP, NOP, NOP, NOP, NOP, NOP, NOP, NOP, NOP`
  - `setuid(0) + execve("/bin/sh")`
Caveats

- Our provided shellcode requires an **executable buffer**
- **What if the buffer is relocated** on every new run?

| Start addr of buffer = ?????
| ---
| Padding to reach RetAddr

WHERE?
Defense: ASLR

- **Address Space Layout Randomization**
  - One of the most common defenses today

- Changes **location of stack** on each execution
  - As well as other memory areas (the heap, libc, etc.)

- Makes buffer overflows significantly harder
  - Can’t “hardcode” address of buffer’s start
  - ... it changes every time!
Defense: ASLR

- How can we overcome ASLR?
Recap: Stack Growth vs. Filling

Stack grows downwards
- Filled upwards

Higher Memory

Lower Memory

41414141
4141414141414141414141

414141414141414141414141
Recap: Redirection to Buffer

Payload = shellcode + NOPs + &buf

Stack grows downwards
- Filled upwards

RetAddr = &buf

Higher Memory

<table>
<thead>
<tr>
<th>Lower Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>9090909090909090909090</td>
</tr>
<tr>
<td>9090909090909090909090</td>
</tr>
<tr>
<td>shellcode 909090909090</td>
</tr>
</tbody>
</table>

Stefan Nagy
Workaround: NOP Slide!

Payload = NOPs + shellcode + &buf

Stack grows downwards
- Filled upwards

Execution moves upwards
- Lower to higher instructions
Workaround: NOP Slide!

Payload = NOPs + shellcode + (&buf + 50)

Stack grows downwards
- Filled upwards

Execution moves upwards
- Lower to higher instructions
- NOP slide leverages this!
Workaround: NOP Slide!

We can’t reliably guess the buffer’s start—it changes every execution!

Payload = NOPs + shellcode + (&buf + 50)
Workaround: NOP Slide!

We can’t reliably guess the buffer’s start—it **changes** every execution!

But, if we prepended our shellcode with a huge **NOP slide**, jumping to the middle of it it will “slide” to our **shellcode**!
Suppose the buffer is **sufficiently large**
- We can still place our shellcode there
- Prepend it with a ton of **NOPs**
Suppose the buffer is **sufficiently large**
- We can still place our shellcode there
- Prepend it with a ton of **NOPs**

- We cannot know buffer’s **exact start**...
Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**

- We cannot know buffer’s **exact start**...
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all

```
setuid(0) + execve("/bin/sh")
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
```
Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**

- We cannot know buffer’s **exact start**...
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all

- **Idea:** spam “**guessed**” buffer addr up the stack

```none
setuid(0) + execve("/bin/sh")
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
```
Suppose the buffer is **sufficiently large**
- We can still place our shellcode there
- Prepend it with a ton of **NOPs**

We cannot know buffer’s **exact start**...
- But we can **guess an address inside of it**
  - It is a really large buffer, after all

**Idea:** spam **“guessed” buffer addr** up the stack
- Eventually we’ll overwrite some **return address**
Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**

- We cannot know buffer’s **exact start**...
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all

- **Idea**: spam “**guessed**” buffer addr up the stack
  - Eventually we’ll overwrite some **return address**
  - When that function returns, jump inside buffer
  - Hit the huge **NOP sled → BOOM!**
Questions?
Application Defense: Data Execution Prevention
Caveats

- Our provided shellcode requires an **executable buffer**

<table>
<thead>
<tr>
<th>Start addr of buffer</th>
<th>Padding to reach RetAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>NOP, NOP, NOP, NOP, NOP, NOP, NOP</strong></td>
</tr>
<tr>
<td></td>
<td><strong>NOP, NOP, NOP, NOP, NOP, NOP, NOP</strong></td>
</tr>
<tr>
<td>setuid(0) + execve(&quot;/bin/sh&quot;)</td>
<td></td>
</tr>
</tbody>
</table>
Caveats

- Our provided shellcode requires an **executable buffer**
- What if the buffer is **prohibited** from being executable?

<table>
<thead>
<tr>
<th>Start addr of buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padding to reach RetAddr</td>
</tr>
<tr>
<td><strong>NOPE</strong></td>
</tr>
</tbody>
</table>
Defense: DEP

- Data Execution Prevention
  - Aka Non-eXecutable (NX) Stack
  - Another common defense seen today

- Attacker can’t execute code on stack
  - Mark pages as EITHER (never both)
  - Read OR write (stack/heap)
  - Executable (.text/code segments)

- Challenges:
  - Self-modifying code, JIT compilation
  - Requires hardware support (MMU/MPU)
Defeating DEP

- Suppose we can still overwrite buffer
  - We **cannot** place our shellcode there
  - But, we can **overwrite other stack items**

- Suppose the program calls a function that can **execute arbitrary commands**
  - execve()
  - system()

```c
main:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    pushl "/bin/ls"
    call system
    leave
    ret
```
Dangerous Calls

- Why are functions like `execve()` and `system()` considered dangerous?
Dangerous Calls

Why are functions like `execve()` and `system()` considered dangerous?

Use of the `system()` function can result in exploitable vulnerabilities, in the worst case allowing execution of arbitrary system commands. Situations in which calls to `system()` have high risk include the following:

- When passing an unsanitized or improperly sanitized command string originating from a tainted source
- If a command is specified without a path name and the command processor path name resolution mechanism is accessible to an attacker
- If a relative path to an executable is specified and control over the current working directory is accessible to an attacker
- If the specified executable program can be spoofed by an attacker

Do not invoke a command processor via `system()` or equivalent functions to execute a command.
Defeating DEP by Controlling Arguments

- Suppose we can still overwrite buffer
  - We **cannot** place our shellcode there
  - But, we can **overwrite other stack items**

- Suppose the program calls a function that can **execute arbitrary commands**
  - execve()
  - system()

- **Idea #1**: overwrite argument to system()
  - Replace it with our shell command ("/bin/sh")

```
main:
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
pushl   "/bin/ls"
call    system
leave
ret
```

Address of "/bin/ls"

system()’s ret addr

Buffer (non-executable)
Defeating DEP by Controlling Arguments

- Suppose we can still overwrite buffer
  - We cannot place our shellcode there
  - But, we can overwrite other stack items

- Suppose the program calls a function that can execute arbitrary commands
  - execve()
  - system()

- Idea #1: overwrite argument to system()
  - Replace it with our shell command ("/bin/sh")
  - Will now execute system("/bin/sh")!

```
main:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
pushl "/bin/ls"
call system
leave
ret
```

Address of "/bin/sh"

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Deceiving DEP by Controlling Arguments
- Suppose we can still overwrite buffer
  - We cannot place our shellcode there
  - But, we can overwrite other stack items
- Suppose the program calls a function that can execute arbitrary commands
  - execve()
  - system()
- Idea #1: overwrite argument to system()
  - Replace it with our shell command ("/bin/sh")
  - Will now execute system("/bin/sh")!

```
main:
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
pushl   "/bin/ls"
call    system
leave
ret
```

Address of system()'s ret addr

```
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
01111111 01110000
00101001 11001010
```

Defeating DEP by Controlling Arguments
Suppose system() isn’t executed, but a call to it exists somewhere

- You can examine the `objdump` to look for “interesting” functions in the program
Defeating DEP via Code Reuse

- Suppose `system()` isn’t executed, but a call to it exists somewhere
  - You can examine the `objdump` to look for “interesting” functions in the program

```c
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, ‘A’, 255);
    buf[255] = ‘\x00’;
    foo(buf);
}
```

- Previous frame ptr
  - `AAAAA...\0`
- `foo()`’s first arg
- `foo()`’s return addr
- `main()`’s frame ptr
- Buffer (non-executable)
Idea #2: create a “fake” call frame for `system()` with our desired arg
Defeating DEP via Code Reuse

- **Idea #2**: create a “fake” call frame for `system()` with our desired arg
### Idea #2: create a "fake" call frame for `system()` with our desired arg

**Diagram:**

- **Previous frame ptr**
  - AAAAAAAAA...\0
- `foo()`'s first arg
- `foo()`'s return addr
- `main()`'s frame ptr
- Buffer (non-executable)

- **Previous frame ptr**
  - AAAAAAAAA...\0
- `foo()`'s first arg
- `foo()`'s return addr
- AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  - AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Defeating DEP via Code Reuse

- **Idea #2:** create a "fake" call frame for `system()` with our desired arg
Defeating DEP via Code Reuse

- **Idea #2:** create a “fake” call frame for `system()` with our desired arg
Defeating DEP via Code Reuse

- **Idea #2**: create a “fake” call frame for `system()` with our desired arg

The diagram shows a comparison between two call frames:

**Left Frame:**
- Previous frame ptr
- 
  
  AAAAAAAAAAAAA...\0
- `foo()`’s first arg
- `foo()`’s return addr
- `main()`’s frame ptr
- Buffer (non-executable)

**Right Frame:**
- Previous frame ptr
- `system()`’s first arg
- `system()`’s return addr
- Address of `system()`
  - AAAAAAAAAAAAAAAAAAAAAAAAAAAA
  - AAAAAAAAAAAAAAAAAAAAAAAAAAAA
  - AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Defeating DEP via Code Reuse

Idea #2: create a “fake” call frame for `system()` with our desired arg

Argument to `system()` is the address of string “/bin/sh”

<table>
<thead>
<tr>
<th>previous frame ptr</th>
<th>string “/bin/sh”</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAAAA...\0</td>
<td>Address of “/bin/sh”</td>
</tr>
<tr>
<td>foo()'s first arg</td>
<td>system()'s return addr</td>
</tr>
<tr>
<td>foo()'s return addr</td>
<td>Address of “/bin/sh”</td>
</tr>
<tr>
<td>main()'s frame ptr</td>
<td>address of string “/bin/sh”</td>
</tr>
<tr>
<td>Buffer (non-executable)</td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
<tr>
<td></td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
<tr>
<td></td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
<tr>
<td></td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
</tbody>
</table>
Defeating DEP via Code Reuse

- Idea #2: create a “fake” call frame for `system()` with our desired arg

---

previous frame ptr

- AAAAAAAAAA...\0

foo()'s first arg

- foo()'s return addr

main()'s frame ptr

- Buffer (non-executable)

Argument to `system()` is the address of string “/bin/sh”

Possible locations: inside the .DATA section, or just the stack!
Defeating DEP via Code Reuse

- **Idea #2:** create a “fake” call frame for `system()` with our desired arg

```
<table>
<thead>
<tr>
<th>previous frame ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAAAA...\0</td>
</tr>
<tr>
<td>foo()'s first arg</td>
</tr>
<tr>
<td>foo()'s return addr</td>
</tr>
<tr>
<td>main()'s frame ptr</td>
</tr>
<tr>
<td>Buffer (non-executable)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>string “/bin/sh”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of “/bin/sh”</td>
</tr>
<tr>
<td>system()'s return addr</td>
</tr>
<tr>
<td>Address of system()</td>
</tr>
<tr>
<td>AAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
<tr>
<td>AAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
<tr>
<td>AAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
</tbody>
</table>
```
Defeating DEP via Code Reuse

- Idea #2: create a "fake" call frame for `system()` with our desired arg

- Address of `/bin/sh`
- Address of `system()`
- `/bin/sh`'s return addr
- `system()`'s return addr
- `/bin/sh`'s first arg
- `foo()`'s return addr
- `foo()`'s frame ptr
- main()'s frame ptr
- Buffer (non-executable)
Defeating DEP via Code Reuse

- **Idea #2:** Create a "fake" call frame for `system()` with our desired arg

<table>
<thead>
<tr>
<th>previous frame ptr</th>
<th>string &quot;/bin/sh&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAA...\0</td>
<td>Address of &quot;/bin/sh&quot;</td>
</tr>
<tr>
<td>foo()'s first arg</td>
<td></td>
</tr>
<tr>
<td>foo()'s return addr</td>
<td>system()'s return addr</td>
</tr>
<tr>
<td>main()'s frame ptr</td>
<td>Address of <code>system()</code></td>
</tr>
<tr>
<td>Buffer (non-executable)</td>
<td></td>
</tr>
</tbody>
</table>

What happens if `system()`'s return address is **overwritten**?
What happens to our exploit when `system()` returns?

- It crashes! 0%
- It executes normally... 0%
- None of the above 0%
Idea #2: create a “fake” call frame for `system()` with our desired arg

What happens if `system()`’s return address is overwritten?

segmentation fault. (Core dumped)
Idea #2: create a "fake" call frame for \texttt{system()} with our desired arg

\begin{itemize}
  \item\textbf{Description}
  
  The function \texttt{\_exit()} terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, \texttt{init}, and the process's parent is sent a \texttt{SIGCHLD} signal.

  The value \texttt{status} is returned to the parent process as the process's exit status, and can be collected using one of the \texttt{wait(2)} family of calls.

  The function \texttt{\_Exit()} is equivalent to \texttt{\_exit()}.
\end{itemize}
Idea #2: create a “fake” call frame for `system()` with our desired arg
Idea #2: create a "fake" call frame for `system()` with our desired arg

- Previous frame ptr
- `foo()`'s first arg
- `foo()`'s return addr
- `main()`’s frame ptr
- Buffer (non-executable)

"/bin/sh"

Address of "/bin/sh"

Address of `_exit()`

Address of `system()`

Defeating DEP via Code Reuse... stealthily!
Questions?
Other Attacks
Return Oriented Programming (ROP)

- Don’t have to jump only to function starts
  - Can jump in the middle of any code
  - x86 has variable instruction lengths
    - Most sequences of “bytes” can be an instruction

- **Idea:** Construct *Turing-complete set of “gadgets”* out of program’s code

- Use **Return-to-libc** like chaining to execute multiple gadgets in sequence!

- ROP is hard to master—we will not expect you to solve this
  - But you can for extra credit ;)

Stefan Nagy
Other Exploitation Techniques

- **1997**
  - Function ptr hijacking
  - Ret-2-Libc attacks
- **1996**
  - Stack overflows
- **1972**
  - First known overflows
- **1997**
  - Heap overflows
- **1998**
  - StackGuard bypasses
- **2005**
  - Ret oriented programming
  - Hardware DEP bypasses
  - ASLR bypasses
  - Integer overflows
  - Format strings
- **2002**
  - Heap grooming
  - Null pointer dereference
  - Double frees
  - Heap spraying
  - JIT spraying
  - ASLR spraying
- **2007**
  - StackGuard bypasses
  - Ret oriented programming
  - Hardware DEP bypasses
  - Integer overflows
  - Format strings
- **2005**
  - Heap overflows
  - Null pointer dereference
  - Double frees
  - Heap spraying
  - JIT spraying
  - ASLR spraying
- **2009**
  - StackGuard bypasses
  - Ret oriented programming
  - Hardware DEP bypasses
  - Integer overflows
  - Format strings
- **2010**
  - Heap overflows
  - Null pointer dereference
  - Double frees
  - Heap spraying
  - JIT spraying
  - ASLR spraying
- **2014**
  - StackGuard bypasses
  - Ret oriented programming
  - Hardware DEP bypasses
  - Integer overflows
  - Format strings
- **2015**
  - Heap overflows
  - Null pointer dereference
  - Double frees
  - Heap spraying
  - JIT spraying
  - ASLR spraying
- **2016**
  - StackGuard bypasses
  - Ret oriented programming
  - Hardware DEP bypasses
  - Integer overflows
  - Format strings
- **2017**
  - Heap overflows
  - Null pointer dereference
  - Double frees
  - Heap spraying
  - JIT spraying
  - ASLR spraying
- **2018**
  - StackGuard bypasses
  - Ret oriented programming
  - Hardware DEP bypasses
  - Integer overflows
  - Format strings
- **2019**
  - Heap overflows
  - Null pointer dereference
  - Double frees
  - Heap spraying
  - JIT spraying
  - ASLR spraying
- **2020**
  - StackGuard bypasses
  - Ret oriented programming
  - Hardware DEP bypasses
  - Integer overflows
  - Format strings
- **2021**
  - Heap overflows
  - Null pointer dereference
  - Double frees
  - Heap spraying
  - JIT spraying
  - ASLR spraying
- **2021**
  - Zero-click exploits
  - Data oriented programming
  - Call oriented programming
  - JMP oriented programming

What's next?
Attack Resources

- Aleph One’s “Smashing the Stack for Fun and Profit”
  - [http://insecure.org/stf/smashstack.html](http://insecure.org/stf/smashstack.html)

- Paul Makowski’s “Smashing the Stack in 2011”

- Blexim’s “Basic Integer Overflows”

- Return-to-libc demo:
  - [http://www.securitytube.net/video/258](http://www.securitytube.net/video/258)
Other Defenses
**Basic idea:** place a **value** near the buffer, check at runtime if it’s **overwritten**
- Analogous to the real-world concept of “canary in a coalmine”

![Stack Canary Diagram]

- RetAddr
- Saved EBP
- Stack Canary
- buf[100]
Stack Canaries

- **Basic idea:** place a **value** near the buffer, check at runtime if it’s **overwritten**
  - Analogous to the real-world concept of “canary in a coalmine”

![Diagram showing the placement of values in the stack]

<table>
<thead>
<tr>
<th>RetAddr</th>
<th>Saved EBP</th>
<th>Stack Canary</th>
<th>buf[100]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>90909090</td>
<td>90909090</td>
</tr>
<tr>
<td>&amp;buf</td>
<td></td>
<td>90909090</td>
<td>9090909090</td>
</tr>
</tbody>
</table>

9090 shellcode
**Basic idea:** place a **value** near the buffer, check at runtime if it’s **overwritten**

- Analogous to the real-world concept of “canary in a coalmine”
Application-level Changes

- Memory error detectors (e.g., AddressSanitizer)
  - **Key idea:** inject “red zones” before and after all memory objects
  - Force a crash when accessing a red zone
  - Catch all subtle (non-crashing) corruptions
  - Implement via instrumentation, custom malloc()
  - **Trade-off:** over 6x execution overhead

![Diagram of memory allocation and usage with red zones]
Application-level Changes

- **Avoiding unsafe functions**

  - **Unsafe:**
    - `strcpy` and friends (`str*`)
    - `sprintf`
    - `gets`

  - **Use instead:**
    - `strncpy` and friends (`strn*`)
    - `snprintf`
    - `fgets`
Preventative Measures

- **Refactoring:**
  - Add bounds checking
  - “Sanitizer” user input

- **Static bug detection tools:**
  - C: Secure Programming Lint
  - C++: CPPCheck

- **Hire CS4440™ graduates**
Preventative Measures

- Refactoring:
  - Add bounds checking
  - “Sanitizer” user input

- Static bug detection tools:
  - C: Secure Programming Lint
  - C++: CPPCheck

- Hire CS4440™ graduates

- Deploy automated testing (next lecture’s topic)
Questions?
Next time on CS 4440...

Automated Bug Finding