CSE 127: Computer Security

Low-level mitigations

Nadia Heninger and Deian Stefan

Some slides adopted from Kirill Levchenko, Stefan Savage, and Stephen Checkoway
Today: mitigating buffer overflows

Lecture objectives:

➤ Understand how to mitigate buffer overflow attacks
➤ Understand the trade-offs of different mitigations
➤ Understand how mitigations can be bypassed
Buffer overflow mitigations

- Avoid unsafe functions (last lecture)
- Stack canaries
  - Separate control stack
  - Memory writable or executable, not both ($W^X$)
  - Address space layout randomization (ASLR)
Stack canaries (again)

• **Goal:** Prevent control flow hijacking by detecting stack-buffer overflows

• **Idea:**
  ➤ Place canary between local variables and saved frame pointer (and return address)
  ➤ Check canary before jumping to return address

• **Approach:**
  ➤ Modify function prologues and epilogues
Example (at a high level)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    func(0xaaaaaaaa, 0xbbbbbbbbbb, argv[1]);
    return 0;
}
```
Compiled, without canaries

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaa, 0xbbbbbbbb, argv[1]);
    return 0;
}
With -fstack-protector-strong

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void foo() {
    printf("hello all!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    func(0xaaaaaaa, 0xbbbbbbb, argv[1]);
    return 0;
}
```

```assembly
func(int, int, char*):
    pushl %ebp
    movl %esp, %ebp
    subl $40, %esp
    movl 16(%ebp), %eax
    movl %eax, -28(%ebp)
    movl $gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    movl $-559038737, -20(%ebp)
    subl $8, %esp
    pushl -28(%ebp)
    leal -16(%ebp), %eax
    pushl %eax
    call strcpy
    addl $16, %esp
    nop
    movl -12(%ebp), %eax
    xorl $gs:20, %eax
    je .L3
    call __stack_chk_fail
.L3:
    leave
    ret
```
With `-fstack-protector-strong`

write canary from `%gs:20` to stack `-12(%ebp)`

compare canary in `%gs:20` to that on stack `-12(%ebp)`

```
func(int, int, char*):
    pushl %ebp
    movl %esp, %ebp
    subl $40, %esp
    movl 16(%ebp), %eax
    movl %eax, -28(%ebp)
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    movl $-559038737, -20(%ebp)
    subl $8, %esp
    pushl -28(%ebp)
    leal -16(%ebp), %eax
    pushl %eax
    call strcpy
    addl $16, %esp
    nop
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L3
    call __stack_chk_fail

.L3:
    leave
    ret
```
Trade-offs

- **Easy to deploy:** Can implement mitigation as compiler pass (i.e., don’t need to change your code)

- **Performance:** Every protected function is more expensive

```asm
.func(int, int, char*)
push $ebp
movi $esp, $ebp
subi $40, $esp
movi 16($ebp), $eax
movi $eax, -$24($ebp)
movi $esp, $eax
movi $eax, -$12($ebp)
xori $eax, $eax
movi $-559038737, -$24($ebp)
subi $8, $esp
push -28($ebp)
leal -$16($ebp), $eax
push $eax
call strcpy
addi $16, $esp
nop
leave
ret
.L3:
leave
ret
```

No stack protection

- `fstack-protector-strong`
When do we add canaries?
When do we add canaries?

- `-fstack-protector`
  - Functions with character buffers $\geq$ `ssp-buffer-size` (default is 8)
  - Functions with variable sized `alloca()`s
When do we add canaries?

• -fstack-protector
  ➤ Functions with character buffers $\geq$ ssp-buffer-size (default is 8)
  ➤ Functions with variable sized allocas

• -fstack-protector-strong
  + Functions with local arrays of any size/type
  + Functions that have references to local stack variables
When do we add canaries?

- **-fstack-protector**
  - Functions with character buffers $\geq$ ssp-buffer-size (default is 8)
  - Functions with variable sized `alloca()`s

- **-fstack-protector-strong**
  + Functions with local arrays of any size/type
  + Functions that have references to local stack variables

- **-fstack-protector-all:**
  - All functions!
There is a cost even for same func:

- No stack protection
  -fstack-protector-strong
  -fstack-protector-all

(we’ll see why in just a bit)
How can we defeat canaries?
How can we defeat canaries?

• Assumption: impossible to subvert control flow without corrupting the canary

• Attack vectors
  ➤ Use targeted write gadget (e.g., with format strings)
  ➤ Pointer subterfuge
  ➤ Overwrite function pointer elsewhere on the stack/heap
  ➤ memcpy buffer overflow with fixed canary
  ➤ Learn the canary
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

int i = 42;

void func(char *str) {
    int *ptr = &i;
    int val = 44;
    char buf[4];
    strcpy(buf,str);
    *ptr = val;
}

int main(int argc, char**argv) {
    func(argv[1]);
    return 0;
}
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

int i = 42;

void func(char *str) {
    int *ptr = &i;
    int val = 44;
    char buf[4];
    strcpy(buf,str);
    *ptr = val;
}

int main(int argc, char**argv) {
    func(argv[1]);
    return 0;
}
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

int i = 42;

void func(char *str) {
    int *ptr = &i;
    int val = 44;
    char buf[4];
    strcpy(buf,str);
    *ptr = val;
}

int main(int argc, char**argv) {
    func(argv[1]);
    return 0;
}
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

int i = 42;

void func(char *str) {
    int *ptr = &i;
    int val = 44;
    char buf[4];
    strcpy(buf,str);
    *ptr = val;
}

int main(int argc, char**argv) {
    func(argv[1]);
    return 0;
}

0x08049b95:  argv[1]

0xfffffd09c:  0x08049b95

%ebp ← saved ebp

%esp ← canary

%ebp  →  0xfffffd09c
     →  0x08049b95
     →  0x41414141

ptr  ←  val

example3.c
Overwrite function pointer on stack

- Similar to previous example, but overwrite function pointer on stack
  - Tricky: compiler can load it into register before `strcpy()`

```c
void func(char *str) {
    void (*fptr)() = &bar;
    char buf[4];
    strcpy(buf,str);
    fptr();
}
```
Can we do anything about this?

• **Problem:** overflowing local variables can allow attacker to hijack control flow

• **Solution:** some implementations reorder local variables, place buffers closer to canaries vs. lexical order

<table>
<thead>
<tr>
<th>arg</th>
<th>arg</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved ret</td>
<td>saved ret</td>
</tr>
<tr>
<td>saved ebp</td>
<td>saved ebp</td>
</tr>
<tr>
<td><strong>canary</strong></td>
<td><strong>canary</strong></td>
</tr>
<tr>
<td>local var</td>
<td>buf[0-3]</td>
</tr>
<tr>
<td>local var</td>
<td>local var</td>
</tr>
<tr>
<td>buf[0-3]</td>
<td>local var</td>
</tr>
</tbody>
</table>
What about function arguments?
What about function arguments?

• Same problem!

```c
void func(char *str, void (*fptr)()) {
    char buf[4];
    strcpy(buf, str);
    fptr();
}
```

• **Solution:** also copy args to the top of the stack to make overwriting them via local variables less likely
What about function arguments?

- Same problem!

```c
void func(char *str, void (*fptr)()) {
    char buf[4];
    strcpy(buf, str);
    fptr()
}
```

- **Solution:** also copy args to the top of the stack to make overwriting them via local variables less likely

<table>
<thead>
<tr>
<th>arg</th>
<th>saved ret</th>
<th>saved ebp</th>
<th>canary</th>
<th>local var</th>
<th>local var</th>
<th>buf[0-3]</th>
</tr>
</thead>
</table>
What about function arguments?

- Same problem!

```c
void func(char *str, void (*fptr)()) {
    char buf[4];
    strcpy(buf,str);
    fptr()
}
```

- **Solution:** also copy args to the top of the stack to make overwriting them via local variables less likely.

<table>
<thead>
<tr>
<th>arg</th>
<th>arg</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved ret</td>
<td>saved ret</td>
</tr>
<tr>
<td>saved ebp</td>
<td>saved ebp</td>
</tr>
<tr>
<td>canary</td>
<td>canary</td>
</tr>
<tr>
<td>local var</td>
<td>local var</td>
</tr>
<tr>
<td>buf[0-3]</td>
<td>buf[0-3]</td>
</tr>
<tr>
<td>arg</td>
<td>arg</td>
</tr>
</tbody>
</table>
That’s what we were seeing before

No stack protection

-\textit{fstack-protector-strong}

\begin{verbatim}
func(int, int, char*){
pushl %ebp
movl %ebp, %ebp
subl $40, %ebp
movl $-559038737, -12(%ebp)
subl $8, %ebp
pushl $16(%ebp)
leal -16(%ebp), %eax
pushl %eax
addl $16, %ebp
nop
movl -12(%ebp), %eax
xorl %gs:0, %eax
je .L3
call _stack_chk_fail
}
.L3:
leave
ret
\end{verbatim}

-\textit{fstack-protector-all}

\begin{verbatim}
pushl %ebp
movl %ebp, %ebp
subl $40, %ebp
movl $-559038737, -12(%ebp)
subl $8, %ebp
pushl $16(%ebp)
leal -16(%ebp), %eax
pushl %eax
addl $16, %ebp
nop
movl -12(%ebp), %eax
xorl %gs:0, %eax
je .L4
call _stack_chk_fail
.L4:
leave
ret
\end{verbatim}
-fstack-protector-strong

```asm
func(int, int, char*):
  pushl %ebp
  movl %esp, %ebp
  subl $40, %esp
  movl 8(%ebp), %eax
  movl %eax, -28(%ebp)
  movl 12(%ebp), %eax
  movl %eax, -32(%ebp)
  movl 16(%ebp), %eax
  movl %eax, -36(%ebp)
  movl %gs:20, %eax
  movl %eax, -12(%ebp)
  xorl %eax, %eax
  movl $-559038737, -20(%ebp)
  subl $8, %esp
  pushl -36(%ebp)
  leal -16(%ebp), %eax
  pushl %eax
  call strcpy
  addl $16, %esp
  nop
  movl -12(%ebp), %eax
  xorl %gs:20, %eax
  je .L4
  call __stack_chk_fail

.L4:
  leave
  ret
```
-fstack-protector-strong

```
func(int, int, char*):
  pushl %ebp
  movl %esp, %ebp
  subl $40, %esp
  movl 8(%ebp), %eax
  movl %eax, -28(%ebp)
  movl 12(%ebp), %eax
  movl %eax, -32(%ebp)
  movl %eax, -32(%ebp)
  movl %eax, -36(%ebp)
  movl %gs:20, %eax
  movl %eax, -12(%ebp)
  xorl %eax, %eax
  movl $-559038737L, -20(%ebp)
  subl $8, %esp
  pushl -36(%ebp)
  leal -16(%ebp), %eax
  pushl %eax
  call strcpy
  addl $16, %esp
  nop
  movl -12(%ebp), %eax
  xorl %gs:20, %eax
  je .L4
  call __stack_chk_fail
.L4:
  leave
  ret
```
-fstack-protector-strong

```assembly
func(int, int, char*):
    pushl  %ebp
    movl  %esp, %ebp
    subl  $40, %esp

    movl  8(%ebp), %eax
    movl  %eax, -28(%ebp)

    movl  12(%ebp), %eax
    movl  %eax, -32(%ebp)

    movl  16(%ebp), %eax
    movl  %eax, -36(%ebp)
    movl  %gs:20, %eax
    movl  %eax, -12(%ebp)
    xorl  %eax, %eax
    movl  $-559038737, -20(%ebp)

    subl  $8, %esp
    pushl  -36(%ebp)
    leal  -16(%ebp), %eax
    pushl  %eax
    call  strcpy
    addl  $16, %esp

    nop
    movl  -12(%ebp), %eax
    xorl  %gs:20, %eax
    je   .L4
    call  __stack_chk_fail

.L4:
    leave
    ret
```
-fstack-protector-strong

```asm
func(int, int, char*):
    pushl %ebp
    movl %esp, %ebp
    subl $40, %esp
    movl %ebp, %eax
    movl %eax, -28(%ebp)
    movl %ebp, %eax
    movl %eax, -32(%ebp)
    movl %ebp, %eax
    movl %eax, -36(%ebp)
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    movl $-559038737, -20(%ebp)
    subl $8, %esp
    pushl -36(%ebp)
    leal -16(%ebp), %eax
    pushl %eax
    call strcpy
    addl $16, %esp
    nop
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L4
    call _stack_chk_fail

.L4:
    leave
    ret
```
-fstack-protector-strong

```c
func(int, int, char*):
    pushl  %ebp
    movl  %esp, %ebp
    subl  $40, %esp
    movl  8(%ebp), %eax
    movl  %eax, -28(%ebp)
    movl  12(%ebp), %eax
    movl  %eax, -32(%ebp)
    movl  16(%ebp), %eax
    movl  %eax, -36(%ebp)
    movl  %gs:20, %eax
    movl  %eax, -12(%ebp)
    xorl  %eax, %eax
    movl  -$559038737, -20(%ebp)
    subl  $8, %esp
    pushl  -36(%ebp)
    leal  -16(%ebp), %eax
    pushl  %eax
    call  strcpy
    addl  $16, %esp

    nop
    movl  -12(%ebp), %eax
    xorl  %gs:20, %eax
    je   .L4
    call  ___stack_chk_fail

.L4:
    leave
    ret
```
How can we defeat canaries?

- Assumption: impossible to subvert control flow without corrupting the canary
- Ideas?
  - Use targeted write (e.g., with format strings)
  - Pointer subterfuge
  - Overwrite function pointer elsewhere on the stack/heap
  - memcpy buffer overflow with fixed canary
  - Learn the canary
memcpy with fixed canary

- Canary values like 0x000d0aff (0, CR, NL, -1) are designed to terminate string ops like strcpy and gets
- Even random canaries have null bytes
- How do we defeat this?
  - Find memcpy/memmove/read vulnerability
How can we defeat canaries?

• Assumption: impossible to subvert control flow without corrupting the canary

• Ideas?
  ➤ Use targeted write (e.g., with format strings)
  ➤ Pointer subterfuge
  ➤ Overwrite function pointer elsewhere on the stack/heap
  ➤ memcpy buffer overflow with fixed canary

Learn the canary
Learn the canary

• Approach 1: chained vulnerabilities
  ➤ Exploit one vulnerability to read the value of the canary
  ➤ Exploit a second to perform stack buffer overflow

• Modern exploits chain multiple vulnerabilities
  ➤ Recent Chinese gov iPhone exploit: 14 vulns!
Learn the canary

• Approach 2: brute force servers (e.g., Apache2)
  ➤ Main server process:
    ➤ Establish listening socket
    ➤ Fork several workers: if any die, fork new one!
  ➤ Worker process:
    ➤ Accept connection on listening socket & process request
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

| saved ret | %ebp | \text{0xbadcaffe} |
| saved ebp | %esp | buf[0-3] |
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>Saved Ret</th>
<th>Saved EBP</th>
<th>EBP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xbadcaffe</td>
<td>%ebp</td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x41414141</td>
<td>%esp</td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Perfect for brute forcing

• Forked process has same memory layout and contents as parent, including canary values!

• The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
<th>saved ebp</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xbadcaffe</td>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
</tbody>
</table>

✓
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

we know size of buffer!
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
<th>saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0xbadcaf41</td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>0x41414141</td>
</tr>
</tbody>
</table>
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
<th>saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xbadcaf41</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x41414141</td>
</tr>
<tr>
<td>%esp</td>
<td>0x41414141</td>
</tr>
</tbody>
</table>

💀 ⚔️
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
<th>saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbadcaf42</td>
<td></td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
</tr>
</tbody>
</table>

%ebp —> 0xbadcaf42
%esp —> 0x41414141
• Forked process has same memory layout and contents as parent, including canary values!

• The fork on crash lets us try different canary values
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
<th>saved ebp</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbadcaffe</td>
<td>0x41414141</td>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
</tbody>
</table>
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved ebp</td>
</tr>
<tr>
<td>%ebp 0xbadc41fe</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%esp 0x41414141</th>
</tr>
</thead>
</table>
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

| saved ret |  
| saved ebp |  
| %ebp      | 0xbadc41fe  
| %esp      | 0x41414141  
|           | 0x41414141  
|           | 0x41414141  

[Diagram showing stack with %esp and %ebp values and skull symbol]
Perfect for brute forcing

• Forked process has same memory layout and contents as parent, including canary values!
• The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
<th>saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%ebp</td>
</tr>
<tr>
<td></td>
<td>0xbadcaffe</td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
</tr>
<tr>
<td>0x41414141</td>
<td></td>
</tr>
<tr>
<td>0x41414141</td>
<td>%esp</td>
</tr>
</tbody>
</table>
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
<th>saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbadcaffe</td>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
</tbody>
</table>
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th></th>
<th>saved ret</th>
<th>saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0xbadcaffe</td>
<td>0x41414141</td>
</tr>
<tr>
<td>%esp</td>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
</tbody>
</table>
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values

<table>
<thead>
<tr>
<th>saved ret</th>
<th>saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbadcaffe</td>
<td>0x41414141</td>
</tr>
</tbody>
</table>

%ebp → 0xbadcaffe
%esp → 0x41414141
Buffer overflow mitigations

- Avoid unsafe functions (last lecture)
- Stack canaries
- Separate control stack
- Memory writable or executable, not both ($W^X$)
- Address space layout randomization (ASLR)
Separate control stack

**Problem:** The stack smashing attacks take advantage of the weird machine: control data is stored next to user data.

**Solution:** Make it less weird by bridging the implementation and abstraction gap: separate the control stack.
Separate control stack

**Problem:** The stack smashing attacks take advantage of the weird machine: control data is stored next to user data

**Solution:** Make it less weird by bridging the implementation and abstraction gap: separate the control stack

![User stack diagram]

- %ebp
- %esp

<table>
<thead>
<tr>
<th></th>
<th>arg i+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg i</td>
<td></td>
</tr>
<tr>
<td>local 1</td>
<td></td>
</tr>
<tr>
<td>local 2</td>
<td></td>
</tr>
</tbody>
</table>
Separate control stack

**Problem:** The stack smashing attacks take advantage of the weird machine: control data is stored next to user data.

**Solution:** Make it less weird by bridging the implementation and abstraction gap: separate the control stack.

---

**User stack**

<table>
<thead>
<tr>
<th></th>
<th>arg i+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg i</td>
<td></td>
</tr>
<tr>
<td>local 1</td>
<td></td>
</tr>
<tr>
<td>local 2</td>
<td></td>
</tr>
</tbody>
</table>

**Control stack**

<table>
<thead>
<tr>
<th></th>
<th>saved ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved ebp</td>
<td></td>
</tr>
</tbody>
</table>

%esp → %ebp

%esp → %esp’
Separate control stack
Separate control stack

• WebAssembly (Wasm) has a separate stack
  ➢ At the Wasm layer: can’t read or manipulate control stack
Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can’t read or manipulate control stack
  - How can we defeat this?
Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can’t read or manipulate control stack
  - How can we defeat this?
- By construction: can’t express stack smashing in Wasm
Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can’t read or manipulate control stack
  - How can we defeat this?
- By construction: can’t express stack smashing in Wasm
  - Challenge: we need to compile C/C++ to Wasm
  - How do we compile buffers, &var, and function ptrs?
Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can’t read or manipulate control stack
  - How can we defeat this?
- By construction: can’t express stack smashing in Wasm
  - Challenge: we need to compile C/C++ to Wasm
  - How do we compile buffers, &var, and function_ptrs?
    - Put them on user stack!
  - So? C programs compiled to Wasm: overwrite function pointers!
Separate control stack

Wasm is not special.

Other byte codes and languages are similar: compiling C to X will inevitably preserve some of C’s bugs.
“SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack. The safe stack stores return addresses, register spills, and local variables that are always accessed in a safe way, while the unsafe stack stores everything else. This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack.”
How do we implement these?

• There is no actual separate stack, we only have linear memory and loads/store instructions

• Put the safe/separate stack in a random place in the address space
  ➤ Assumption: location of control/stack stack is secret
  ➤ How do we defeat this?
Intel’s upcoming shadow stack

- Addresses both the performance and security issues
  - New shadow stack pointer (%ssp)
    - call and ret automatically update %esp and %ssp
    - Can’t update shadow stack manually
  - May need to rewrite code that manipulates stack manually

```
<table>
<thead>
<tr>
<th>%ebp</th>
<th>arg i+1</th>
<th>arg i</th>
<th>saved ret</th>
<th>saved ebp</th>
<th>local var</th>
<th>buf</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>saved ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

▶️ %ssp
How do we defeat this?

Find a function pointer and overwrite it to point to shellcode!
Buffer overflow mitigations

- Avoid unsafe functions (last lecture)
- Stack canaries
- Separate control stack
- Memory writable or executable, not both ($W^X$)
- Address space layout randomization (ASLR)
W^X: write XOR execute

- **Goal:** prevent execution of shell code from the stack
- **Insight:** use memory page permission bits
  - Use MMU to ensure memory cannot be both writeable and executable at same time
- Many names for same idea:
  - XN: eXecute Never
  - W^X: Write XOR eXecute
  - DEP: Data Execution Prevention
Recall our memory layout
Recall our memory layout

- kernel
- user stack
- shared libs
- runtime heap
- static data segment
- text segment
- unused

Permissions:
- rw
- rx
Recall our memory layout

- kernel
- user stack
- shared libs
- runtime heap
- static data segment
- text segment
- unused

- saved ret
- saved ebp
- buf[0-3]

- %ebp
- %esp
Recall our memory layout
Recall our memory layout
**W^X tradeoffs**

- **Easy to deploy:** No code changes or recompilation
- **Fast:** Enforced in hardware
  - Also a downside: what do you do on embedded devices?
- What if some pages need to be both writeable and executable?
  - What programs do you use that need this?
How can we defeat W^X?

- Can still write to stack
  - Jump to existing code
- Search executable for code that does what you want
  - E.g. if program calls `system("/bin/sh")` you’re done
  - libc is a good source of code (return-into-libc attacks)
Employees must wash hands before returning to libc
Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string “/bin/sh”

Our vulnerable function:

<table>
<thead>
<tr>
<th>saved ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved ebp</td>
</tr>
<tr>
<td>buf[4-7]</td>
</tr>
<tr>
<td>buf[0-3]</td>
</tr>
</tbody>
</table>

%esp →
Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string 
  "/bin/sh"
Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string “/bin/sh”
We already did this with foo

Calling \texttt{system()} is the same, but need to argument to string \texttt{"/bin/sh"}
Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string “/bin/sh”
Can we inject code?
Can we inject code?

The function `mprotect()` changes the access protections for the calling process's memory pages containing any part of the address range in the interval `[addr, addr+len-1]`. `addr` must be aligned to a page boundary.

If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a `SIGSEGV` signal for the process.

`prot` is a combination of the following access flags: `PROT_NONE` or a bitwise-or of the other values in the following list:

- `PROT_NONE` The memory cannot be accessed at all.
- `PROT_READ` The memory can be read.
- `PROT_WRITE` The memory can be modified.
- `PROT_EXEC` The memory can be executed.
Can we inject code?

- Just-in-time compilers produce data that becomes executable code
- JIT spraying:
  - 1. Spray heap with shellcode (and NOP slides)
  - 2. Overflow code pointer to point to spray area
What does JIT shellcode look like?
What does JIT shellcode look like?

```c
var g1 = 0;
...
var g7 = 0;

for (var i=0; i<100000; ++i) {
    g1 = 50011;  \ pop ebx; ret;
    g2 = 50009;  \ pop ecx; ret;
    g3 = 12828721; \ xor eax, eax; ret;
    g4 = 12811696; \ mov 0x7d, al; ret;
    g5 = 12833329; \ xor edx, edx; ret;
    g6 = 12781490; \ mov 0x7, dl; ret;
    g7 = 12812493; \ int 0x80; ret;
}
```
What does JIT shellcode look like?

```javascript
var g1 = 0;
...
var g7 = 0;

for (var i=0; i<100000; ++i) {
  g1 = 50011;  // pop ebx; ret;
  g2 = 50009;  // pop ecx; ret;
  g3 = 12828721;  // xor eax, eax; ret;
  g4 = 12811696;  // mov 0x7d, al; ret;
  g5 = 12833329;  // xor edx, edx; ret;
  g6 = 12781490;  // mov 0x7, dl; ret;
  g7 = 12812493;  // int 0x80; ret;
}
```

The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines

Michalis Athanasakis  Elias Athanasopoulos  Michalis Polytechnisi  Georgios Portokalidis  Sotiris Ioannidis
FORTH, Greece  FORTH, Greece  Stony Brook University  Stevens Institute of Tech.  FORTH, Greece
michal@ics.forth.gr  elhanat@ics.forth.gr  mikp@cs.stonybrook.edu  geporto@stevens.edu  sotiris@ics.forth.gr
How do we defend against this?

• Modify the JavaScript JIT
  ➤ Store JavaScript strings in separate heap from rest
  ➤ Blind constants

• Ongoing arms race
  ➤ E.g., Wasm makes it easier for attackers: gap between Wasm and x86/ARM is much smaller than JavaScript
Buffer overflow mitigations

- Avoid unsafe functions (last lecture)
- Stack canaries
- Separate control stack
- Memory writable or executable, not both ($W^X$)

Address space layout randomization (ASLR)
ASLR

- Traditional exploits need precise addresses
  - stack-based overflows: location of shellcode
  - return-into-libc: library addresses
- **Insight:** Make it harder for attacker to guess location of shellcode/libc by randomizing the address of different memory regions
When do we randomize?
When do we randomize?

```
[debug code master]
$ cat /proc/self/maps | grep '(libc|heap|stack)'
55555555f000-55555555f000 rw-p 00000000 00:00 0 [heap]
7ffff7c90000-7ffff7df3000 r-xp 00000000 fe:02 2100102 /usr/lib/libc-2.28.so
7ffff7d3000-7ffff7f3d000 r-xp 00250000 fe:02 2100102 /usr/lib/libc-2.28.so
7ffff7f3d000-7ffff7f6800 r-xp 00180000 fe:02 2100102 /usr/lib/libc-2.28.so
7ffff7f6800-7ffff7f6700 r-xp 00180000 fe:02 2100102 /usr/lib/libc-2.28.so
7ffff7f6700-7ffff7f5000 r-xp 00180000 fe:02 2100102 /usr/lib/libc-2.28.so
7ffff7f5000-7ffff7f8000 r-xp 00180000 fe:02 2100102 /usr/lib/libc-2.28.so
7fffffda000-7fffffff7f000 r-wp 00000000 00:00 0 [heap]
```

[debug code master]
```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

[debug code master]
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
How much randomness?

32-bit PaX ASLR (x86)

Stack:

Execution: 1 0 1 0 R R R R R R R R R R R R R R R R R R R R R R R R R R R 0 0 0 0

fixed | random (24 bits) | zero

Mapped area:

Execution: 0 1 0 0 R R R R R R R R R R R R R R R R R R R R R R R R R R 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

fixed | random (16 bits) | zero

Executable code, static variables, and heap:

Execution: 0 0 0 0 R R R R R R R R R R R R R R R R R R R R R R R R R R R 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

fixed | random (16 bits) | zero
Tradeoff

• **Intrusive:** Need compiler, linker, loader support
  ➢ Process layout must be randomized
  ➢ Programs must be compiled to not have absolute jumps

• **Incurs overhead:** increases code size & perf overhead

• Also mitigates heap-based overflow attacks
How can we defeat ASLR?

- Older Linux would let local attacker read the stack start address from `/proc/<pid>/stat`
- `-fno-pie` binaries have fixed code and data addresses
  - Enough to carry out control-flow-hijacking attacks
- Each region has random offset, but layout is fixed
  - Single address in a region leaks every address in region
- Brute force for 32-bit binaries and/or pre-fork binaries
- Heap spray for 64-bit binaries
Derandomizing ALSR

- **Attack goal:** call `system()` with attacker arg
- **Target:** Apache daemon
  - **Vulnerability:** buffer overflow in `ap_getline()`

```c
char buf[64];
...
strcpy(buf, s); // overflow
```
Assumptions

- $W^X$ enabled
- PaX ASLR enabled
  - Apache forks child processes to handle client interaction
  - Recall how re-randomization works?
Attack steps

- Stage 1: Find base of mapped region

- Stage 2: Call system() with command string

<table>
<thead>
<tr>
<th>Mapped area:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 R R R R R R R R R R R R 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>fixed</td>
</tr>
</tbody>
</table>

- Stage 2: Call system() with command string
How do we find the mapped

- Observation: layout of mapped region (libc) is fixed
- Overwrite saved return pointer with a guess to `usleep()`
  - base + offset of `usleep`
  - non-negative argument

```
<table>
<thead>
<tr>
<th>ap_getline()</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
</tr>
<tr>
<td>saved ret</td>
</tr>
<tr>
<td>saved ebp</td>
</tr>
<tr>
<td>%ebp</td>
</tr>
<tr>
<td>%esp</td>
</tr>
<tr>
<td>buf</td>
</tr>
</tbody>
</table>
```
How do we find the mapped

• Observation: layout of mapped region (libc) is fixed
• Overwrite saved return pointer with a guess to `usleep()`
  ➤ base + offset of `usleep`
  ➤ non-negative argument

```
0x10101010
0xdeadbeef
~&usleep()
0xdeadbeef
%ebp
buf
%esp
```
Finding base of mapped region

• If we guessed `usleep()` address right
  ➤ Server will freeze for 16 seconds, then crash

• If we guessed `usleep()` address wrong
  ➤ Server will (likely) crash immediately

• Use this to tell if we guessed base of mapped region correctly
Finding base of mapped region

• If we guessed `usleep()` address right
  ➤ Server will freeze for 16 seconds, then crash

• If we guessed `usleep()` address wrong
  ➤

• Use this to tell if we guessed base of mapped region correctly
Finding base of mapped region

- If we guessed `usleep()` address right
  - Server will freeze for 16 seconds, then crash

- If we guessed `usleep()` address wrong
  - Server will (likely) crash immediately

- Use this to tell if we guessed base of mapped region correctly
Derandomizing ASLR
Derandomizing ASLR

- What is the success probability?
Derandomizing ASLR

- What is the success probability?
  - \(1/2^{16}\) — 65,536 tries maximum
Derandomizing ASLR

• What is the success probability?
  ➤ $1/2^6$ — 65,536 tries maximum

• Do we need to derandomize the stack base?
Derandomizing ASLR

• What is the success probability?
  ➤ $1/2^6$ — 65,536 tries maximum

• Do we need to derandomize the stack base?
  ➤ No!
Attack steps

- **Stage 1**: Find base of mapped region (libc)

- **Stage 2**: Call system() with command string

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{fixed} & \text{random} & \text{zero} \\
\hline
0 & 1 & 0 & 0 & R & R & R & R & R & R & R & R & R & R & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

\text{Mapped area:}
How do we call system?

- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`
How do we call system?

- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of `buf` looks like argument to `system()`
- Append address of `system()`

```
&buf
0xdeadbeef
&system
addr of ret
...
addr of ret
0xdeadbeef
"/bin/sh"
%esp
```
How do we call system?

- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`
How do we call system?

• Overwrite saved return pointer with address of `ret` instruction in libc
• Repeat until address of buf looks like argument to `system()`
• Append address of `system()`

```
&buf
0xdeadbeef
&system
addr of ret
...
addr of ret
0xdeadbeef
"/bin/sh"
```
How do we call system?

- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`
How do we call system?

- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`
How do we call system?

- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`
How do we call system?

• Overwrite saved return pointer with address of `ret` instruction in libc
• Repeat until address of `buf` looks like argument to `system()`
• Append address of `system()`
Buffer Overflow Defenses

• Avoid unsafe functions
• Stack canary
• Separate control stack
• Memory writable or executable, not both (W^X)
• Address Space Layout Randomization (ASLR)
None are perfect, but in practice