CSE 127: Introduction to Security

Buffer overflow defenses

Nadia Heninger

UCSD

Winter 2021 Lecture 3

Some slides from Kirill Levchenko, Stefan Savage, Stephen Checkoway, and Deian Stefan
This lecture will be recorded and made available to registered students on Canvas.
Today: mitigating buffer overflows

Lecture objectives:

• Understand how to mitigate buffer overflow attacks
• Understand the tradeoffs of different mitigations
• Understand how mitigations can be bypassed.
Can we just avoid writing C code that has buffer overflow bugs?
Yes! Avoid unsafe functions!

• `strcpy`, `strcat`, `gets`, etc.

• This is a good idea in general...

• Requires manual code rewrite

• Non-library functions may be vulnerable

• e.g. user creates their own `strcpy`

• No guarantee you found everything

• Alternatives are also error-prone!
Yes! Avoid unsafe functions!

• strcpy, strcat, gets, etc.
• This is a good idea in general...
• But...
  • Requires manual code rewrite
  • Non-library functions may be vulnerable
    • e.g. user creates their own strcpy
  • No guarantee you found everything
  • Alternatives are also error-prone!
Even printf is tricky

If buf is under control of attacker, is 

printf("\%s\n", buf)

call safe?
Even printf is tricky

Is printf(buf) safe?
Even printf is tricky

Is printf(“%s\n”) safe?
printf can be used to read and write memory
⇒ control flow hijacking!

Exploiting Format String Vulnerabilities

scut / team teso

September 1, 2001

https://cs155.stanford.edu/papers/formatstring-1.2.pdf
If we can’t avoid writing buccy C code,
Can we prevent or mitigate exploitation?
Buffer overflow mitigations

- Avoid unsafe functions
  → Stack canaries

- Separate control stack

- Memory writable or executable, not both \((W^X)\)

- Address space layout randomization
Miner's canary  [ edit ]

Canaries were used as sentinel species for use in detecting carbon monoxide in coal mining from around 1913 when the idea was suggested by John Scott Haldane. Toxic gases such as carbon monoxide or asphyxiant gases such as methane in the mine would affect the bird before affecting the miners. Signs of distress from the bird indicated to the miners that conditions were unsafe. The birds were generally kept in carriers which had small oxygen bottles attached to revive the birds, so that they could be used multiple times within the mine. The use of miners' canaries in British mines was phased out in 1986.

The phrase "canary in a coal mine" is frequently used to refer to a person or thing which serves as an early warning of a coming crisis. By analogy, the term "climate canary" is used to refer to a species (called an indicator species) that is affected by an environmental danger prior to other species, thus serving as an early warning system for the other species with regard to the danger.
Stack canaries

- Prevent control flow hijacking by detecting 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
High-level example

```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

```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,0xbbbbbbbb,argv[1]);
    return 0;
}
```

For the compiled code:
```
pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $-559038737, -12(%ebp)
subl $8, %esp
pushl 16(%ebp)
lea -16(%ebp), %eax
pushl %eax
call strcpy
addl $16, %esp
nop
leave
ret
```
```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, 0xbbbbbbbb, argv[1]);
    return 0;
}
```

```asm
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
```
Compiled with `-fstack-protector-strong`

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

compare canary in `%gs:20` to value on stack `-12(%ebp)`
Tradeoffs of stack canaries

- **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

```
-fstack-protector-strong
```

No stack protection
Can we defeat stack canaries?

- **Assumption:** Impossible to subvert control flow without corrupting the canary.

- Think outside the box
Can we defeat stack canaries?

• **Assumption:** Impossible to subvert control flow without corrupting the canary.

• Think outside the box
  • Overwrite function pointer elsewhere on the stack/heap
  • Pointer subterfuge
  • 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;
}
```c
#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;
}
```

---

**Pointer subterfuge**

```plaintext
#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;
}
```
# Pointer subterfuge

```
#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;
}
```

![Diagram showing memory layout and pointers](chart.png)
void func(char *str) {
    void (*fptr)() = &bar;
    char buf[4];
    strcpy(buf,str);
    fptr()
}
Or overwrite a function pointer argument

```c
void func(char *str, void (*fptr)()) {
    char buf[4];
    strcpy(buf, str);
    fptr();
}
```
What can we do about this?

• **Problem:** Overflowing locals and arguments can allow attacker to hijack control flow

**Solution:**

• Move buffers closer to canaries vs. lexical order
• Copy args to top of stack
What can we do about this?

• **Problem:** Overflowing locals and arguments can allow attacker to hijack control flow

• **Solution:**
  - Move buffers closer to canaries vs. lexical order
  - Copy args to top of stack
Your compiler does this already

-\texttt{-fstack-protector}

- Functions with char bufs $\geq$ ssp-buffer-size (default=8)

- Functions with variable-sized \texttt{alloca()}s
Your compiler does this already

- `fstack-protector`
  - Functions with char bufs $\geq$ ssp-buffer-size (default=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
Your compiler does this already

- **-fstack-protector**
  - Functions with char bufs \( \geq \) ssp-buffer-size (default=8)
  - Functions with variable-sized `alloca()`

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

- **-fstack-protector-all:**
  - All functions!
Zooming in on our compiled code

```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
```
Can we defeat stack canaries?

• **Assumption:** Impossible to subvert control flow without corrupting the canary.

• Think outside the box
  • Overwrite function pointer elsewhere on the stack/heap
  • Pointer subterfuge
    → memcpy buffer overflow with fixed canary
  • Learn the canary
How do we pick canaries?

• Pick a clever value!
  • E.g. 0x000d0aff (0, CR, NL, -1) to terminate string operations like `strcpy` and `gets`
  • Even if attacker knows value, can’t overwrite past canary!
Not all overflows are due to strings

Many other functions handle buffers

• E.g. memcpy, memmove, read

• These are also error-prone!

```c
void func(char *str) {
    char buf[1024];
    memcpy(buf, str, strlen(str));
}
```
How do we pick canaries?

- Pick a random value!
  - When?
Can we defeat stack canaries?

• **Assumption:** Impossible to subvert control flow without corrupting the canary.

• Think outside the box
  • Overwrite function pointer elsewhere on the stack/heap
  • Pointer subterfuge
  • 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 government iPhone exploit: 14 vulns!
Learn the canary

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

\[
\begin{align*}
%ebp & \rightarrow 0x\text{badcaffe} \\
%esp & \rightarrow \text{buf}[0-3]
\end{align*}
\]
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>0xbadcaffe</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved ebp</td>
<td>0x41414141</td>
</tr>
<tr>
<td>%ebp</td>
<td>0xbadcaffe</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>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0xbadcaffe</td>
<td>0x41414141</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
<tr>
<td></td>
<td></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

Figured out 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
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></td>
<td>0xbdcaf41</td>
<td></td>
</tr>
<tr>
<td></td>
<td></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>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xbadcaf42</td>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
<tr>
<td></td>
<td></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

| saved ret |
| saved ebp |
| %ebp       |
| 0xbadcaf42 |
| 0x41414141 |
| 0x41414141 |

| %esp       |
| 0x41414141 |

連れ去られるとしたら、

骷髅と骨を

骷髅と骨を

骷髅と骨を
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></td>
<td>0xbadcaffe</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
<tr>
<td></td>
<td></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></td>
<td><strong>0xbadcaffe</strong></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
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 showing saved ret, saved ebp, ebp=0xbadc41fe, esp=0x41414141, ebp=0x41414141]
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  →  0xbadcaffe
0x41414141  0x41414141
%esp  →  0x41414141  0x41414141
```
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>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0xbadcaffe</td>
</tr>
<tr>
<td></td>
<td>0x41414141</td>
</tr>
<tr>
<td></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></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

![Canary Values](image)

Figured out size of canary!
Buffer overflow mitigations

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

**Problem:** Control data is stored next to data

**Solution:** Bridge the implementation and abstraction gap: separate the control stack

![Diagram showing user and control stacks](image)
Safe stack

**Problem:** Unsafe data structures stored next to control

**Solution:** Move unsafe data structures to separate stack
How do we implement a separate stack?

• There is no actual separate stack
  • We only have linear memory and load/store instructions

• Put the safe/separate stack in a random place in the address space
  • Location of control/safe stack is secret
How do we defeat this?

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

• Avoid unsafe functions

• Stack canaries

• Separate control stack

→ Memory writable or executable, not both ($W^X$)

• Address space layout randomization
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 executeable at the same time

- Many names for same idea:
  - XN: eXecute Never
  - W^X: Write XOR eXecute
  - DEP: Data Execution Prevention
Recall our memory layout

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

- rw
- rx

- saved ret
- saved ebp
- %ebp
- %esp
- buf[0-3]
Recall our memory layout

- **Kernel**
- **User stack**
- **Shared libs**
- **Runtime heap**
- **Static data segment**
- **Text segment**
- **Unused**

- **Shellcode**
- **Hijacked ret**
- **%ebp**
- **%esp**
Recall our memory layout

- Kernel
- User stack
- Shared libs
- Runtime heap
- Static data segment
- Text segment
- Unused

 RW
 RW
 RW
 RW
 RW

 Shellcode
 Hijacked ret
 %ebp
 %esp
W^X tradeoffs

- **Easy to deploy:** No code changes or recompilation
- **Fast:** Enforced in hardware
  - Downside: What do you do on embedded devices?
- Some pages need to be both writeable and executable
  - Why?
How can we defeat W^X?

- Can still write to return address stored on the 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
Redirecting control flow to `system()`

- We redirected control flow in previous lecture to `foo()`.
- Calling `system()` is the same, but need to have argument string ""/bin/sh"" on stack.
Redirecting control flow to `system()`

- We redirected control flow in previous lecture to `foo()`
- Calling `system()` is the same, but need to have argument string “/bin/sh” on stack.
Redirecting control flow to `system()`
Redirecting control flow to `system()`

- `leave` → `movl %ebp, %esp`
  - `pop %ebp`
- `ret` → `popl %eip`

```
leave     → movl %ebp, %esp
           pop %ebp
ret       → popl %eip
```
Redirecting control flow to `system()`

After `leave`

```
leave → movl %ebp, %esp
        pop %ebp
ret → popl %eip
```
Redirecting control flow to `system()`

After `ret`

- `leave` → `movl %ebp, %esp`
  - `pop %ebp`

- `ret` → `popl %eip`

```
%ebp → ??

%esp → 
  - `&exit`
  - `&system`
    - `????`

%eip → &system
```
To system this looks like a normal call
But I want to execute shellcode, not just call \texttt{system()}!
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 spray area
What does JIT shellcode look like?

```java
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 Polychronakis  Georgios Portokalidis  Sotiris Ioannidis  
FORTH, Greece  FORTH, Greece  Stony Brook University  Stevens Institute of Tech.  FORTH, Greece  
michath@ics.forth.gr  elathan@ics.forth.gr  mikepo@cs.stonybrook.edu  gportoka@stevens.edu  sotiris@ics.forth.gr
Buffer overflow mitigations

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

→ Address space layout randomization (ASLR)
Address Space Layout Randomization (ASLR)

- Traditional exploits need precise addresses
  - stack-based overflows: 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
How much do we randomize?

32-bit PaX ASLR (x86)

**Stack:**

- Fixed: 1010
- Random (24 bits): RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR0000
- Zero

**Mapped area:**

- Fixed: 0100
- Random (16 bits): RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR0000000000000000000000
- Zero

**Executable code, static variables, and heap:**

- Fixed: 0000
- Random (16 bits): RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR0000000000000000000000
- Zero
ASLR Tradeoffs

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

- **Incurs overhead:** Increases code size and performance overhead

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

• 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
Buffer overflow mitigations

- Avoid unsafe functions
- Stack canaries
- Separate control stack
- Memory writable or executable, not both ($W^X$)
- Address space layout randomization (ASLR)
None are perfect, but in practice they raise the bar dramatically.