CSE 127:
Introduction to Security

Buffer overflow attacks and defenses

Nadia Heninger

UCSD

Winter 2023 Lecture 3

Some slides from Kirill Levchenko, Stefan Savage, Stephen

What’s wrong with this program?

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

What’s wrong with this program?

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

MEMCPY(3) Linux Programmer's Manual MEMCPY(3)

NAME top

memcpy - copy memory area

SYNOPSIS top

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

What's wrong with this program?

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

MEMCPY(3) Linux Programmer's Manual MEMCPY(3)

NAME top

memcpy - copy memory area

SYNOPSIS top

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

What’s wrong with this program?

```
void vulnerable(int len = 0xffffffff, char *data) {
  char buf[64];
  if (len = -1 > 64)
    return;
  memcpy(buf, data, len = 0xffffffff);
}
```

What’s wrong with this program?

```
void vulnerable(int len = 0xffffffff, char *data) {
  char buf[64];
  if (len = -1 > 64)
    return;
  memcpy(buf, data, len = 0xffffffff);
}
```

MEMCPY(3) Linux Programmer's Manual MEMCPY(3)

NAME top

memcpy - copy memory area

SYNOPSIS top

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

Let’s fix it

```
void safe(size_t len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

Is this program safe?

```
void f(size_t len, char *data) {
  char *buf = malloc(len+2);
  if (buf == NULL)
    return;
  memcpy(buf, data, len);
  buf[len] = '\n';
  buf[len+1] = '\0';
}
```

Is this program safe?

```
void f(size_t len = 0xffffffff, char *data) {
  char *buf = malloc(len+2 = 0x00000001);
  if (buf == NULL)
    return;
  memcpy(buf, data, len = 0xffffffff);
  buf[len] = ‘\n’;
  buf[len+1] = ‘\0’;
}
```

No!

Three flavors of integer overflows

- Truncation bugs
  - e.g. assigning an `int64_t` into `int32_t`
- Arithmetic overflow bugs
  - e.g. adding huge unsigned numbers
- Sign bugs
  - e.g. treating signed number as unsigned

Still relevant classes of bugs

Issue 952406: Security: Possible OOB related to chrome_sqlite3_malloc
Reported by mlfbr...@stanford.edu on Fri, Apr 12, 2019, 1:59 PM PDT

Code

VULNERABILITY DETAILS
Possible OOB with chrome_sqlite3_malloc

REPRODUCTION CASE
There's a pattern of using sqlite malloc functions that call chrome_sqlite3_malloc in combination with traditional memory operations (e.g., memcpy). There may be invariants that make this ok, or a principle here that I am not aware of. Thanks for your time.

chrome_sqlite3_malloc takes an int size argument, while memcpy takes a size_t size argument. On x86-64 this means that chrome_sqlite_3_malloc's size argument is width 32, while memcpy's is width 64. This can lead to potentially concerning wrapping behavior for extreme allocation sizes (depending on the compiler, optimizations, etc).

For example:

Function fts3UpdateDocTotals
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3399)

(1) a = sqlite3_malloc((sizeof(u32)+10)*nStat);
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3416)
...
(2) memset(a, 0, sizeof(u32)*(nStat));
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3434)

Depending on optimization level etc, this may turn into:

(1)
size = mul i32 nstat 14
chrome_sqlite3_malloc(size)

(2)
tmp = sign extend nstat to i64
size = shl tmp 2
memset(size)

If nstat is a very large i32, the multiplication in step (1) *may* wrap. Nothing in (2) will wrap because of the sign extend, leading to an OOB.

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…

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(buf)` safe?

Even printf is tricky

If `buf` is under control of attacker, is

`printf(‘‘%s\n’’, buf)`

safe?

Even printf is tricky

Is `printf(‘‘%s\n’’)` safe?

`printf` can be used to read and write memory
$\implies$ 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 buggy C code,

Can we prevent or mitigate exploitation?

Buffer overflow mitigations

- Avoid unsafe functions
- $\rightarrow$ 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.[14] Toxic gases such as carbon monoxide or asphyxiant gases such as methane[15] 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.[16] The use of miners' canaries in British mines was phased out in 1986.[17][18]

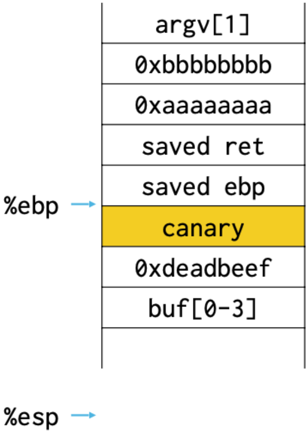
The phrase "canary in a coal mine" is frequently used to refer to a person or thing which serves as an early warning of a coming crisis. By analogy, the term "climate canary" is used to refer to a species (called an indicator species) that is affected by an environmental danger prior to other species, thus serving as an early warning system for the other species with regard to the danger.[19]

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,0xbbbbbbbb,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;
}
```

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $-559038737, -12(%ebp)
        subl    $8, %esp
        pushl   16(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        leave
        ret
```

Compiled 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(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)

```
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
```

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

No stack protection

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $-559038737, -12(%ebp)
        subl    $8, %esp
        pushl   16(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        leave
        ret
```

-fstack-protector-strong

```
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
```

Can we defeat stack canaries?

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

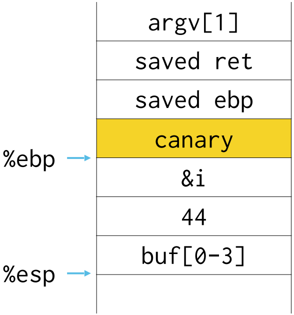
Can we defeat stack canaries?

- **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

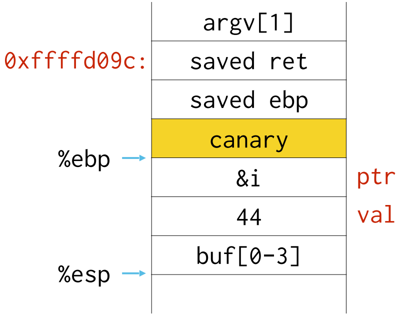
Pointer subterfuge

```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;
}
```



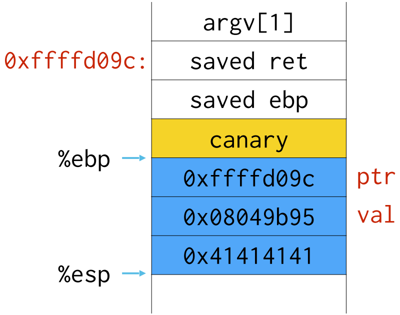
Pointer subterfuge

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

void foo() {
0x08049b95:   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;
}
```



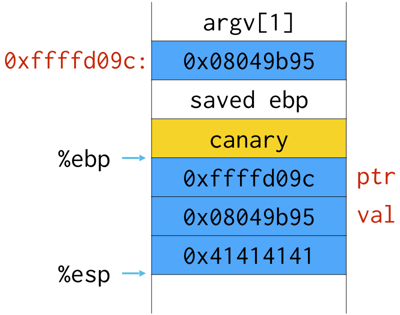
Pointer subterfuge

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

           void foo() {
0x08049b95:  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;
           }
```



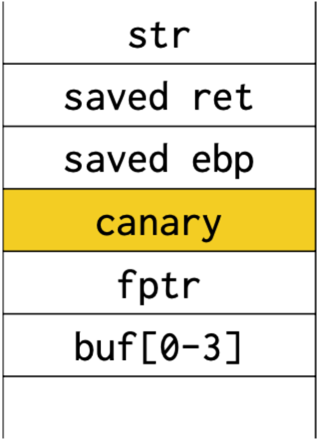
Pointer subterfuge

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

            void foo() {
0x08049b95:   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;
            }
```



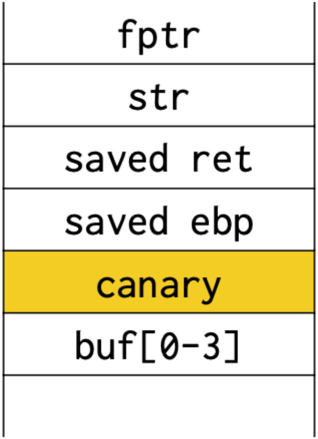
Overwrite function pointer on stack

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



Or overwrite a function pointer argument

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

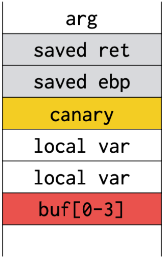


What can we do about this?

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

  Solution:

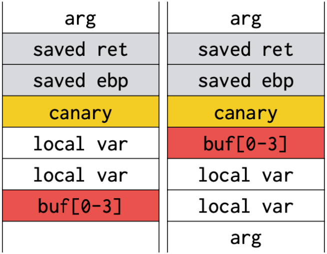
  - Move buffers closer to canaries vs. lexical order
  - Copy args to top of stack



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

-fstack-protector

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

Zooming in on our compiled code

```
               func(int, int, char*):
                       pushl   %ebp
                       movl    %esp, %ebp
                       subl    $40, %esp
copy arg1              movl    8(%ebp), %eax
                       movl    %eax, -28(%ebp)
copy arg2              movl    12(%ebp), %eax
                       movl    %eax, -32(%ebp)
copy arg3              movl    16(%ebp), %eax
                       movl    %eax, -36(%ebp)
write canary           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

    $\rightarrow$ 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!

```
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

  $\rightarrow$ Learn the canary

Learn the canary

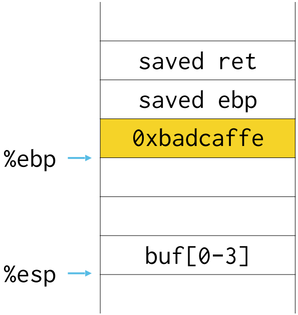
- Approach 1: chained vulnerabilities
  - Exploit one vulnerability to read the value of the canary
  - Exploit a second to perform stack buffer overflow
- Modern exploits chain multiple vulnerabilities
  - Recent Chinese government iPhone exploit: 14 vulns!

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

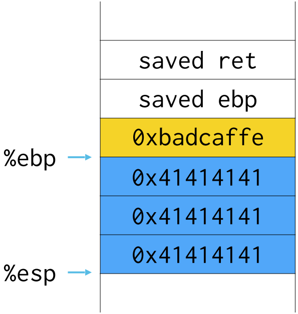
Perfect for brute forcing

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



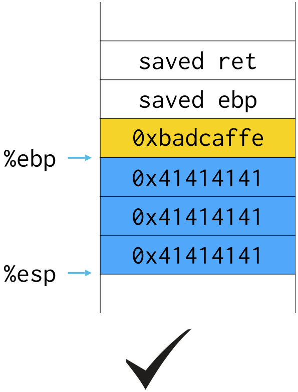
Perfect for brute forcing

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



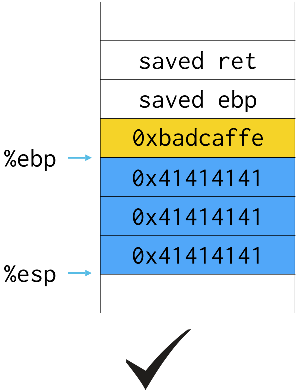
Perfect for brute forcing

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



Perfect for brute forcing

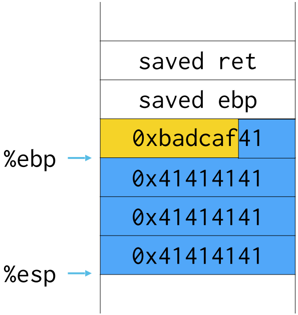
- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



Figured out size of buffer!

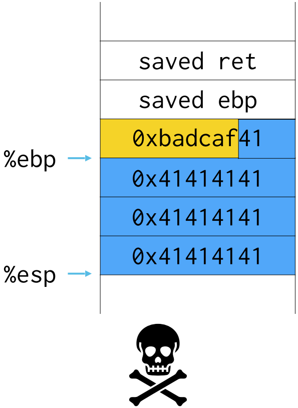
Perfect for brute forcing

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



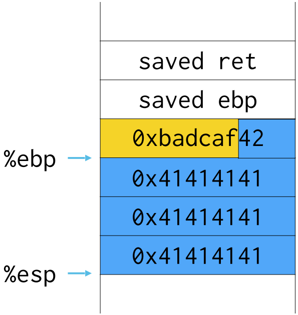
Perfect for brute forcing

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



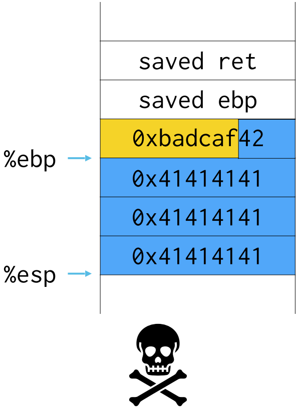
Perfect for brute forcing

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



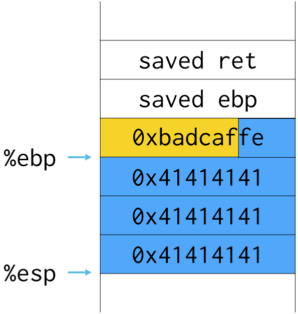
Perfect for brute forcing

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



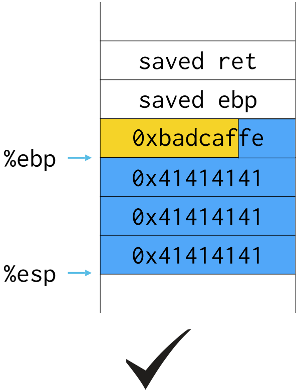
Perfect for brute forcing

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



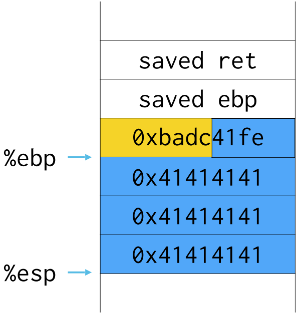
Perfect for brute forcing

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



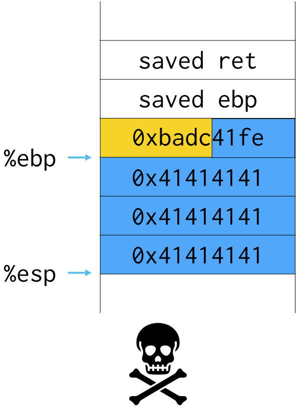
Perfect for brute forcing

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



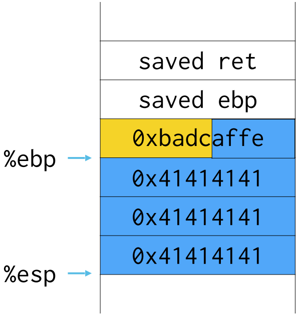
Perfect for brute forcing

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



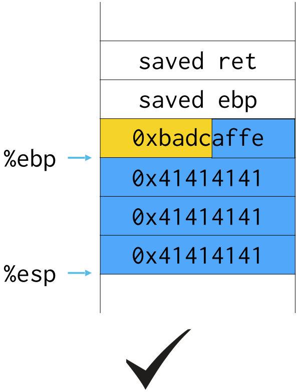
Perfect for brute forcing

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



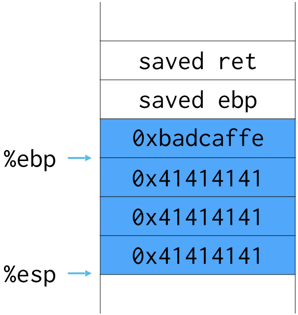
Perfect for brute forcing

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



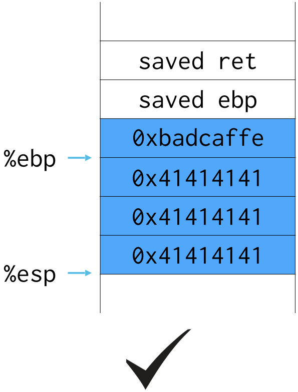
Perfect for brute forcing

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



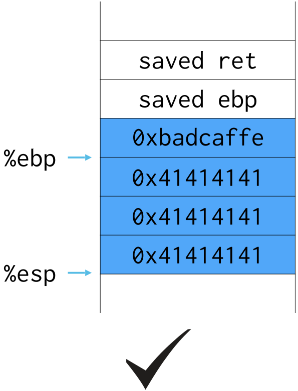
Perfect for brute forcing

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



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 canary!

Buffer overflow mitigations

- Avoid unsafe functions
- Stack canaries

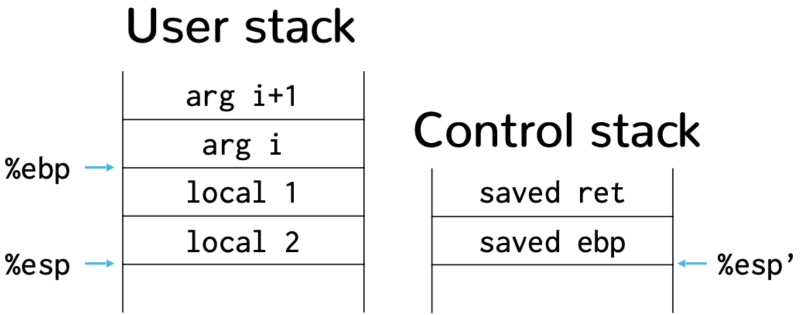
$\rightarrow$ Separate control stack

- Memory writable or executable, not both (W^X)
- Address space layout randomization

Separate control stack

Problem: Control data is stored next to data

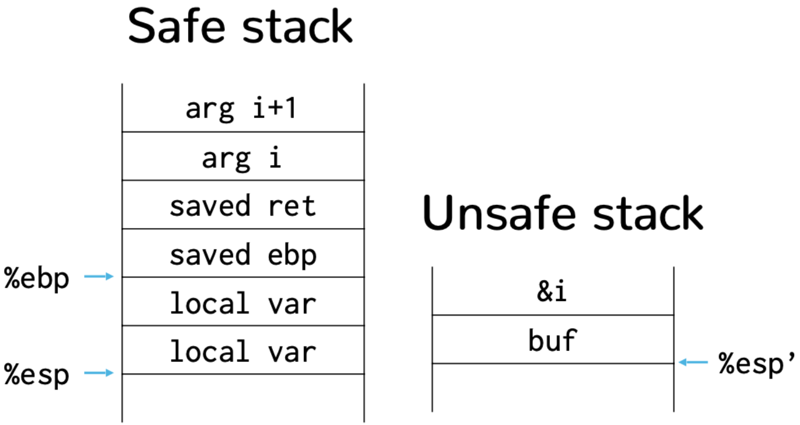
Solution: Bridge the implementation and abstraction gap: separate the control stack



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

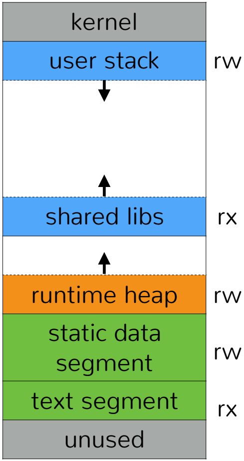
$\rightarrow$ Memory writable or executable, not both (W^X)

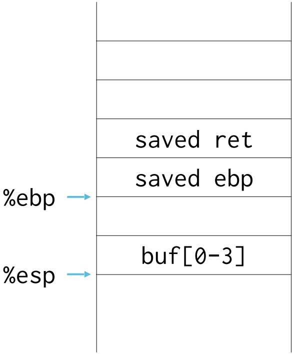
- Address space layout randomization

W^X: write XOR execute

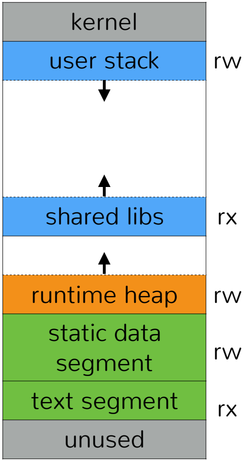
- **Goal:** Prevent execution of shell code from the stack
- **Insight:** Use memory page permission bits
  - Use MMU to ensure memory cannot be both writeable and executeable at the same time
- Many names for same idea:
  - XN: eXecute Never
  - W^X: Write XOR eXecute
  - DEP: Data Execution Prevention

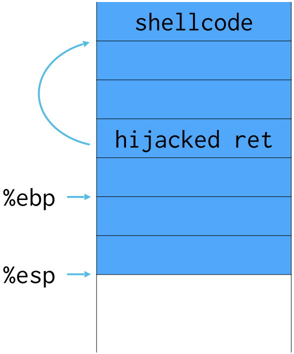
Recall our memory layout



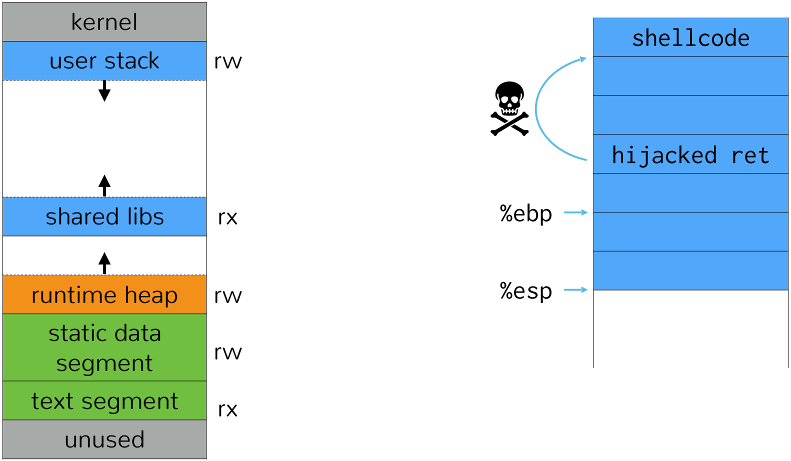


Recall our memory layout





Recall our memory layout



W^X tradeoffs

- **Easy to deploy:** No code changes or recompilation
- **Fast:** Enforced in hardware
  - Downside: What do you do on embedded evices?
- Some pages need to be both writeable and executable
  - Why?

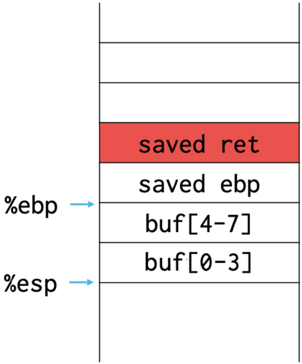
How can we defeat W^X?

- 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)

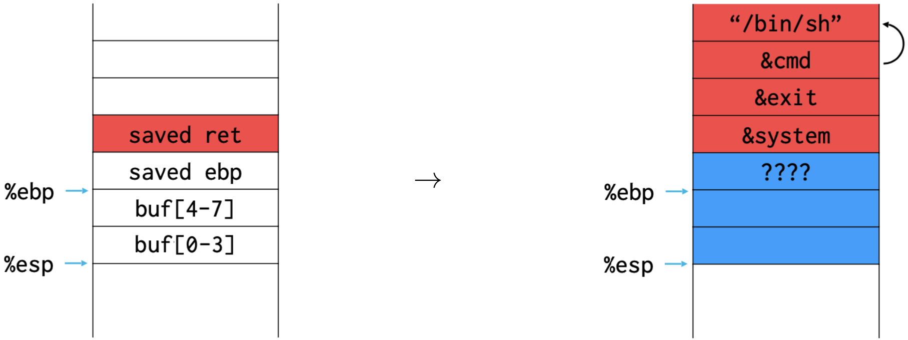

Employees must
wash hands before
returning to libc

Redirecting control flow to `system()`

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

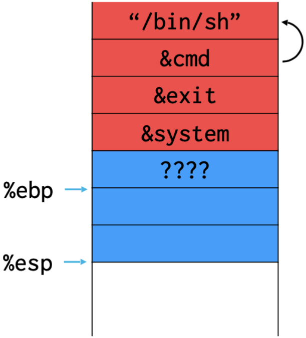


Redirecting control flow to `system()`



Redirecting control flow to `system()`

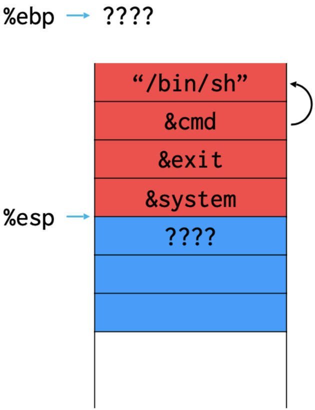
```
leave    → movl %ebp, %esp
           pop %ebp

ret      → popl %eip
```



Redirecting control flow to `system()`

After `leave`

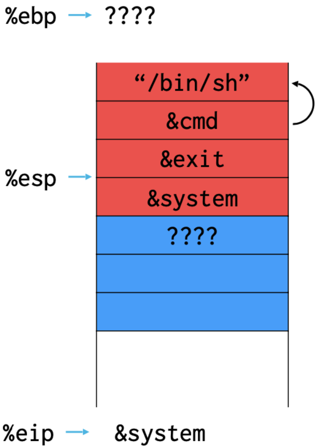
```
leave    →  movl %ebp, %esp
             pop %ebp

ret       →  popl %eip
```



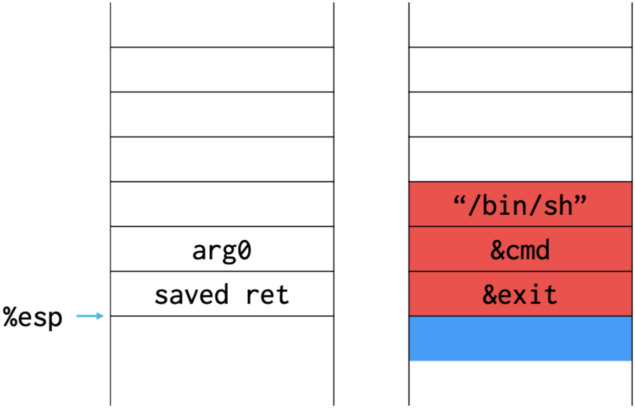
Redirecting control flow to `system()`

After `ret`

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

ret       → popl %eip
```



To system this looks like a normal call



But I want to execute shellcode, not just call `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?

```
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, FORTH, Greece, michath@ics.forth.gr

Elias Athanasopoulos, FORTH, Greece, elathan@ics.forth.gr

Michalis Polychronakis, Stony Brook University, mikepo@cs.stonybrook.edu

Georgios Portokalidis, Stevens Institute of Tech., gportoka@stevens.edu

Sotiris Ioannidis, FORTH, Greece, sotiris@ics.forth.gr

Buffer overflow mitigations

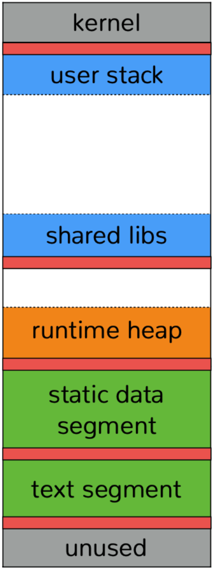
- Avoid unsafe functions
- Stack canaries
- Separate control stack
- Memory writable or executable, not both (W^X)

$\rightarrow$ Address space layout randomization (ASLR)

Address Space Layout Randomization (ASLR)

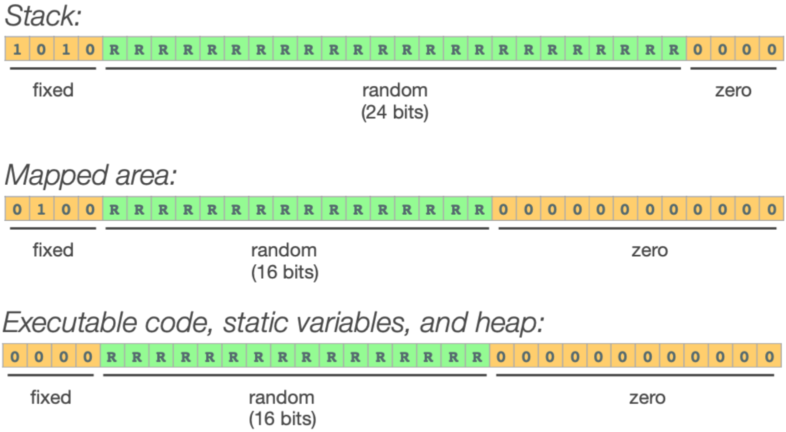
- Traditional exploits need precise addresses
  - stack-based overflows: shellcode
  - return-into-libc: library addresses

- **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)



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

When do we randomize?

Many options.

- At boot?
- At compile/link time?
- At run/load time?
- On fork?

What’s the tradeoff?

How can we defeat ASLR?

- `-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

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.