

# CSE 127 Computer Security

Stefan Savage, Spring 2020, Lecture 5

---

Control Flow Vulnerabilities:  
Defenses and Evolution

# Logistics

---

- Assignment 2 is assigned today
  - You get to write some exploits
- Partners
  - If you are going to use a partner, you need to turn in your partner form by this Friday
  - Remember, this is a one-time choice. Partner now is partner for rest of quarter

# Today: mitigating buffer overflows

## Lecture objectives:

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

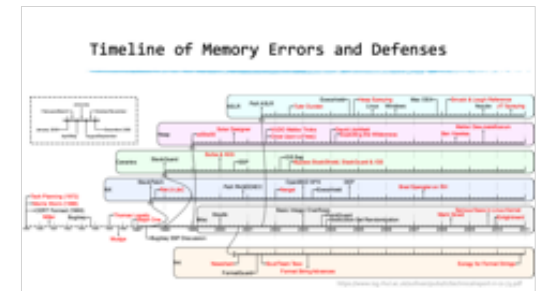
# Countermeasures and Mitigations

---

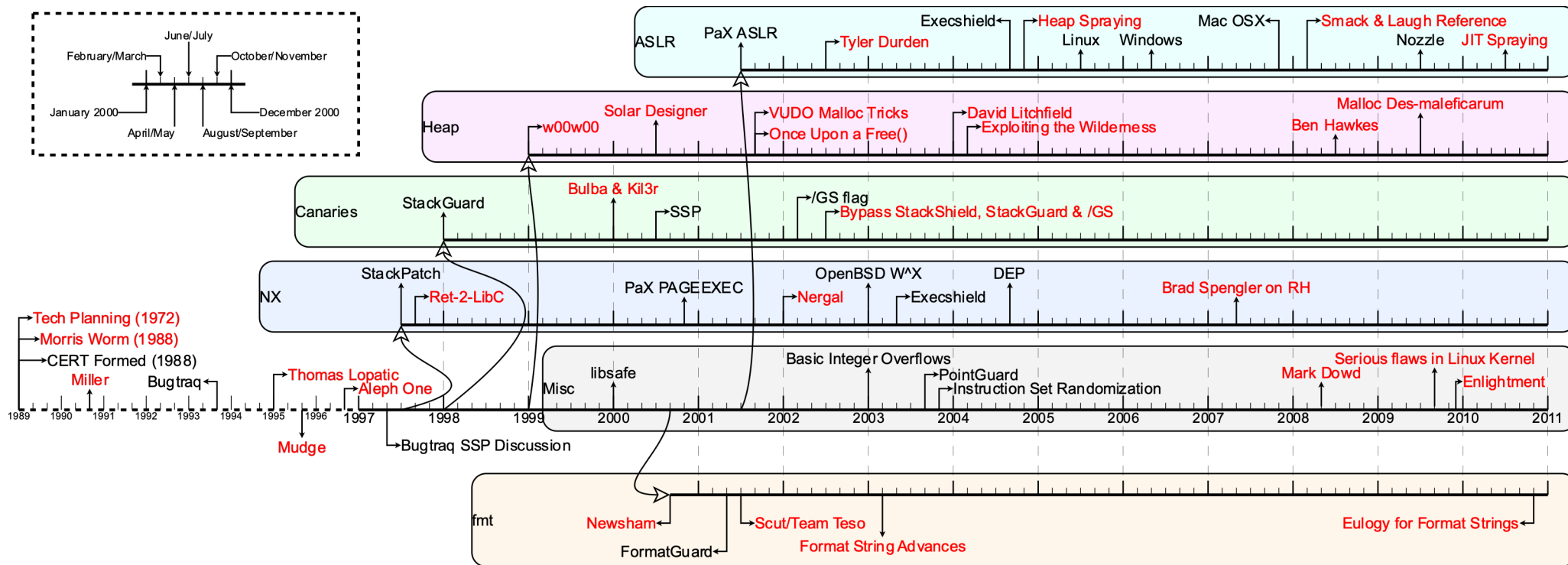
- Asking developers to not insert vulnerabilities has not worked.
  - People will make mistakes and introduce vulnerabilities during development
  - Not all of these vulnerabilities will be discovered prior to release
- As the next line of defense, *countermeasures* are introduced that make reliable exploitation harder or *mitigate* the impact of remaining vulnerabilities.
  - Will **not** stop all exploits
  - Make exploit development more difficult and costly (how much?)
  - Mitigations are important, but it's much better to write code properly!
- Ongoing *arms race* between defenders and attackers
  - Co-evolution of defenses and exploitation techniques

# Countermeasures and Mitigations

- As we consider different mitigations:
  - Challenge assumptions
  - Keep thinking of how you can circumvent each “solution”
- Security is an ongoing arms race
  - Developers introduce new features. Attackers devise ways to exploit these features. Defenders devise new countermeasures or mitigations. Attackers adapt to the new countermeasures, the defenders refine their approach, and the cycle continues with no end in sight...
    - ... and full employment on all sides



# Timeline of Memory Errors and Defenses



# Countermeasures and Mitigations

---

- What do we want to prevent?
  - Overwriting of the return address?
  - Hijacking of the control flow?
  - All out of bounds memory access?
- Approaches we'll look at today
  - Try to detect overwrite of control data
    - **Stack canaries/cookies**
  - Try to make it difficult to redirect control flow to attacker code
    - **Memory protection (DEP/W^X)**
    - **Address Space Layout Randomization (ASLR)**

# Stack Canary

---

## Miner's canary [\[ edit \]](#)

---

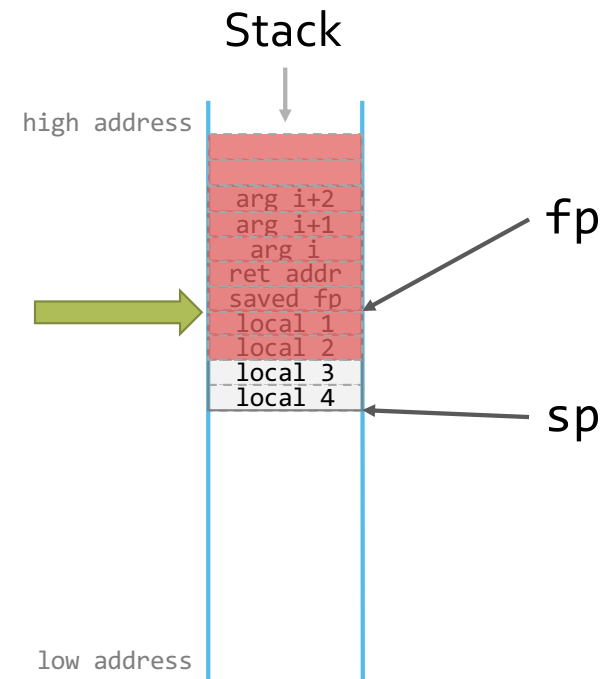
Canaries were used as [sentinel species](#) for use in detecting carbon monoxide in [coal mining](#) from around 1913 when the idea was suggested by [John Scott Haldane](#).<sup>[14]</sup> [Toxic gases](#) such as [carbon monoxide](#) or [asphyxiant gases](#) such as [methane](#)<sup>[15]</sup> in the 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.<sup>[16]</sup> The use of miners' canaries in [British](#) mines was phased out in 1986.<sup>[17][18]</sup>

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.<sup>[19]</sup>



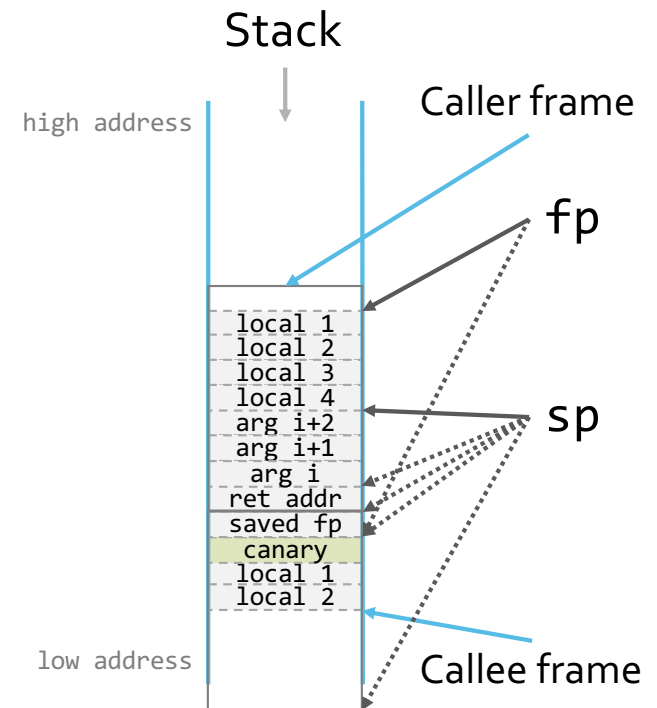
# Stack Canary

- Stack-based buffer overflows
  - Typical attacks overflow local buffer into control data (i.e., ret)
- **Detect** overwriting of the return address
  - Place a special value (aka *canary* or cookie) between local variables and the saved frame pointer
  - Check that value before popping saved frame pointer and return address from the stack



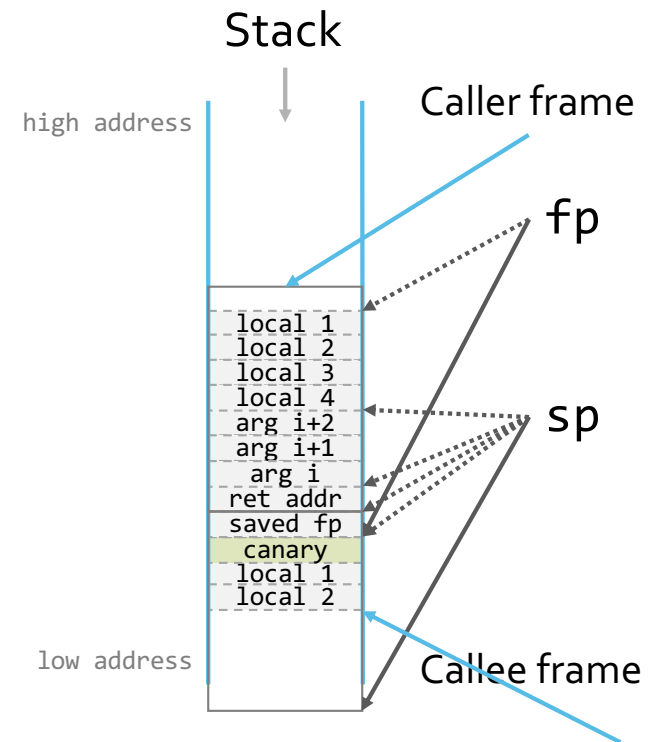
# Stack Canary

- Calling a function
  - Caller
    - Pass arguments
    - Call and save return address
  - Callee
    - Save old frame pointer
    - Set frame pointer = stack pointer
    - **Allocate stack space for local storage + space for the canary**
    - **Push canary**



# Stack Canary

- When returning
  - Callee
    - Check canary against a global 'gold' copy
      - Jump to exception handler if different
    - Pop local storage
      - Set stack pointer = frame pointer
    - Pop frame pointer
    - Pop return address and return
  - Caller
    - Pop arguments



# Stack Canary

---

- What value should we use for the canary?
- What about `0x000A0DFF`?
  - **Terminator Canary**
  - Hard to insert via string functions
- What's the problem with using a fixed value?
- Other options?
- Rather than making canaries hard-to-insert, we can make them hard-to-guess.
  - **Random canaries** are secure as long as they remain secret.

`0x000A0DFF`

```
-----  
| | | | -> EOF  
| | | ---> LF  
| | -----> CR  
| -----> NUL
```

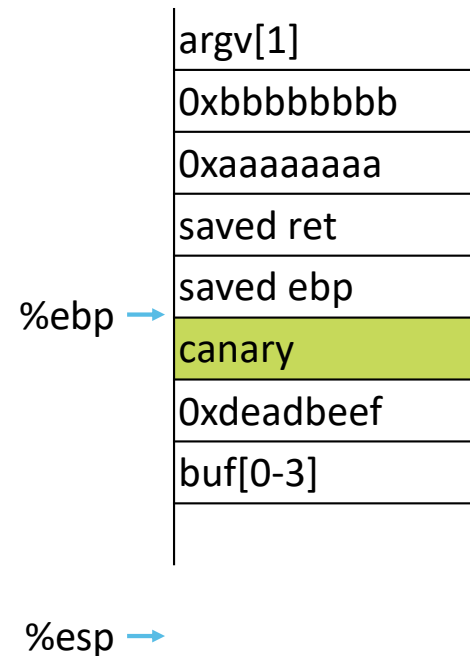
## Example (at a high level)

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

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

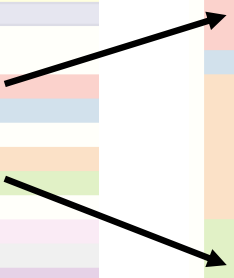
void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

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



# Compiled, without canaries

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void foo() {
6      printf("hello all!!\n");
7      exit(0);
8  }
9
10 void func(int a, int b, char *str) {
11     int c = 0xdeadbeef;
12     char buf[4];
13     strcpy(buf, str);
14 }
15
16 int main(int argc, char**argv) {
17     func(0xaaaaaaaa, 0xbbbbbbbb, argv[1]);
18     return 0;
19 }
```



```
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
```

# With -fstack-protector-strong

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void foo() {
6      printf("hello all!!\n");
7      exit(0);
8  }
9
10 void func(int a, int b, char *str) {
11     int c = 0xdeadbeef;
12     char buf[4];
13     strcpy(buf, str);
14 }
15
16 int main(int argc, char**argv) {
17     func(0xaaaaaaaa, 0xbbbbbbbb, argv[1]);
18     return 0;
19 }
```

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)

```
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
```

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



# 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

## 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
```

## When do we add canaries?

- -fstack-protector
  - Functions with character buffers  $\geq$  ssp-buffer-size (default is 8)
  - Functions with variable sized alloca()s (dynamic allocation on stack)
- -fstack-protector-strong
  - Functions with local arrays of any size/type
  - Functions that have references to local stack variables
- -fstack-protector-all:
  - All functions!

# More protection can increase cost for every function

## 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
```

## -fstack-protector-all

```
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
```

(we'll see why in just a bit)

# Stack Canary Limitations

---

- How can stack canaries be bypassed?
  - Assumption: impossible to subvert control flow without corrupting the canary.
  - Challenge it!
  - Is it possible to overwrite the canary with a valid canary value?
  - Can you cause trouble by overwriting non-protected data?
  - Is it possible to overwrite critical data without overwriting the canary?

# Stack Canary Bypasses

---

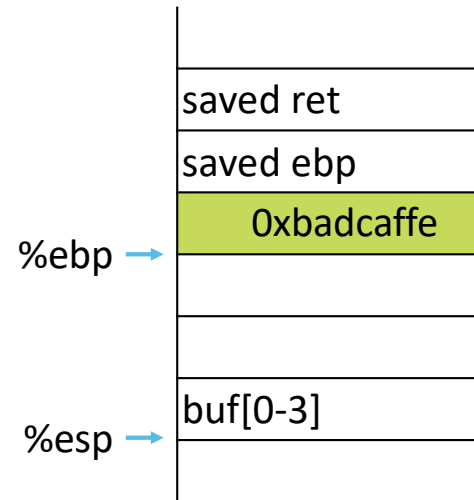
- Can an attacker overwrite the canary with the correct value?
  - Are terminator canaries impossible to insert?
    - With string functions, yes... but other attacks: length-bound loops, `memcpy()`, etc.
  - How random are random canaries? Can an attacker guess them?
    - Canary value is selected at process creation, stays the same for life of process
    - Network services that fork child processes to handle connections.
      - What happens if you guess wrong? **How many guesses do you need?**

# Learning the canary

- Example brute forcing servers (e.g., Apache2)
  - Main server process:
    - Establish listening socket on the network
    - Fork several workers: if any workers die, fork new one!
  - Worker process:
    - Accept connection on listening socket & process request

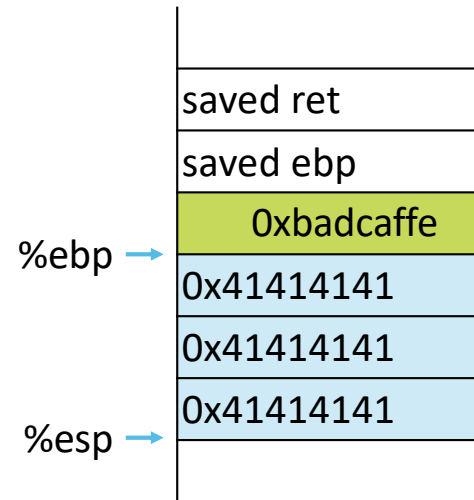
## Perfect for brute forcing

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



## 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

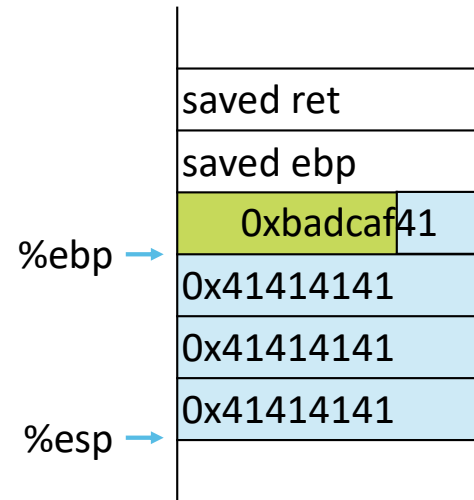


we know size of buffer!



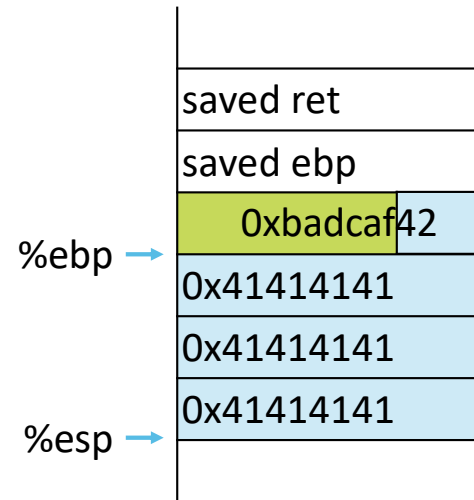
## Perfect for brute forcing

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



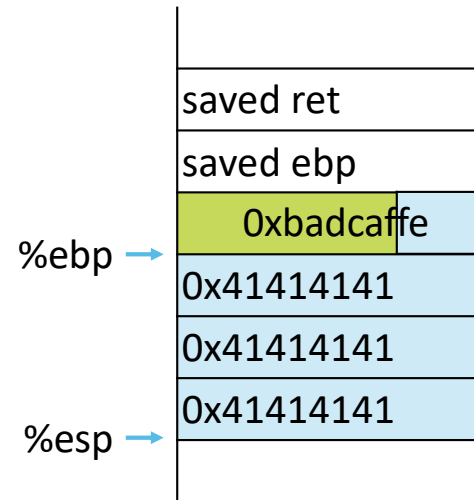
## Perfect for brute forcing

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



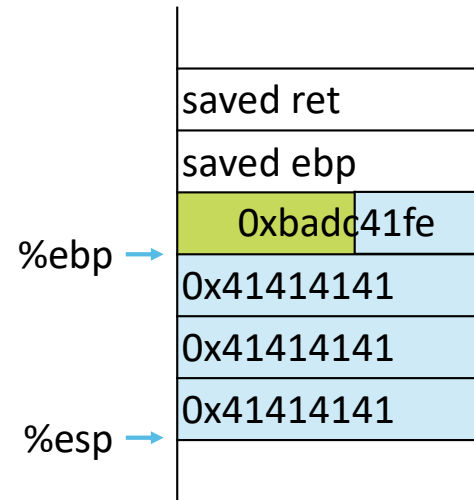
## Perfect for brute forcing

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



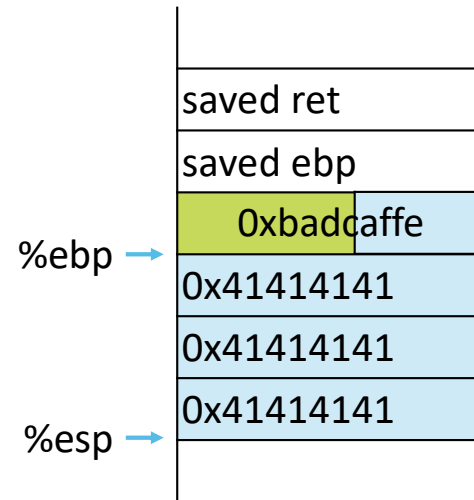
# Perfect for brute forcing

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



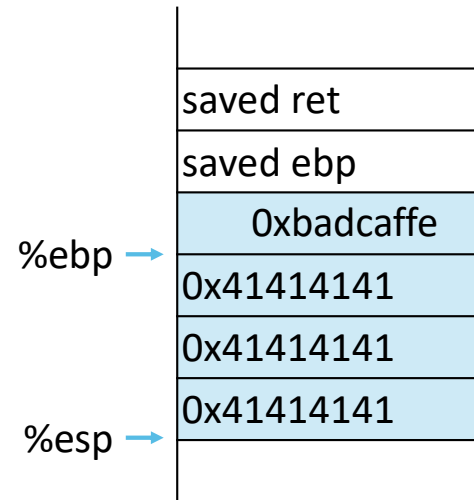
## Perfect for brute forcing

- 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



# Stack Canary Bypasses

---

- Can an attacker overwrite the canary with the correct value?
  - Are terminator canaries impossible to insert?
    - With string functions, yes... but other attacks: length-bound loops, `memcpy()`, etc.
  - How random are random canaries? Can an attacker guess them?
    - Canary value is selected at process creation, stays the same for life of process
    - Network services that fork child processes to handle connections.
      - What happens if you guess wrong? **How many guesses do you need?**
  - How secret are random canaries? Can an attacker leak them?
    - An **information leak** might disclose the value of the canary

# Stack canary bypass via information leak

---

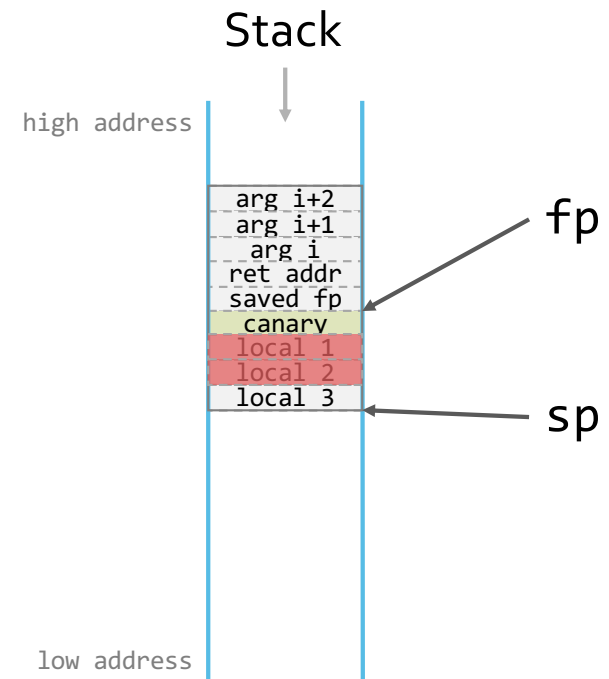
- Reading outside the bounds can also be a security issue.
- If the read data is returned to the attacker, it could disclose sensitive information and allow further exploitation.
  - E.g, printf vuln
- “Chaining” multiple vulnerabilities is common for modern exploits





# Stack Canary Bypasses

- Can an attacker overwrite something that is not protected by canaries?
  - Local variables
    - Variables that store result of a security check
      - Eg. isAuthenticated, isValid, isAdmin, etc.
    - Variables used in security checks
      - Eg. buffer\_size, etc.
    - Data pointers ([pointer subterfuge](#))
      - Potential for further memory corruption
    - Function pointers
      - Direct transfer of control when function is called through overwritten pointer
  - Exception control data



# Pointer subterfuge

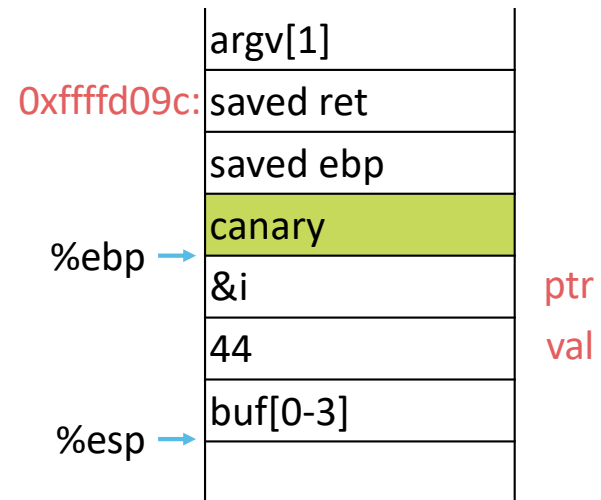
```
#include <stdio.h>
#include <string.h>

0x08049b95: 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

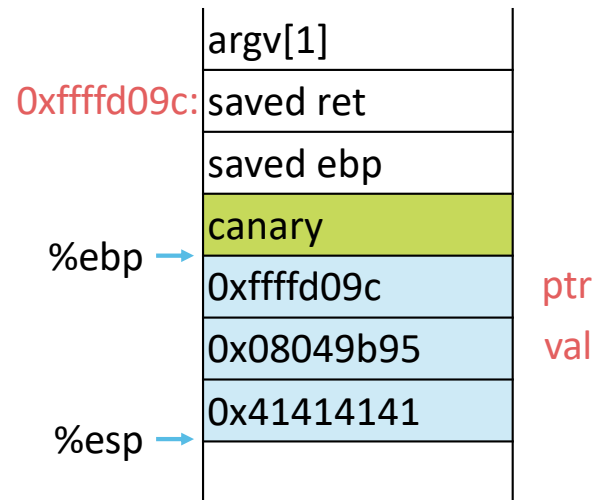
```
#include <stdio.h>
#include <string.h>

0x08049b95: 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

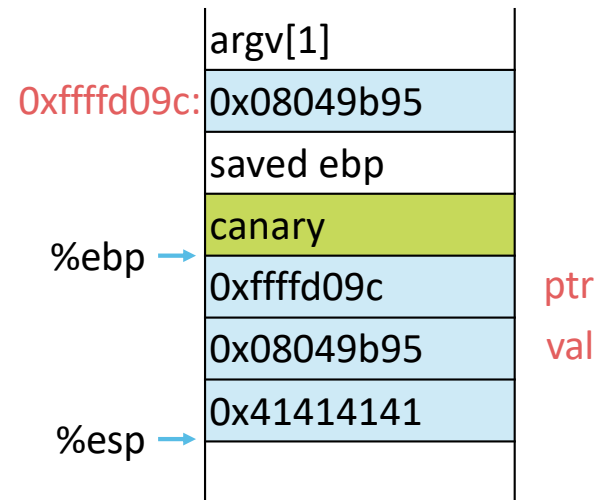
```
#include <stdio.h>
#include <string.h>

0x08049b95: 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;
}
```



# Defending against Stack Canary bypasses

---

- **Problem:** overflowing local variables can allow attacker to hijack control flow
- **Partial Solution:** some implementations reorder local variables, place buffers closer to canaries vs. lexical order

|           |           |
|-----------|-----------|
| arg       | arg       |
| saved ret | saved ret |
| saved ebp | saved ebp |
| canary    | canary    |
| local var | buf[0-3]  |
| local var | local var |
| buf[0-3]  | local var |
|           |           |

# Defending against Stack Canary Bypasses

- What about function arguments?
- Same problem!

```
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

|           |           |
|-----------|-----------|
| arg       | arg       |
| saved ret | saved ret |
| saved ebp | saved ebp |
| canary    | canary    |
| local var | local var |
| local var | local var |
| buf[0-3]  | buf[0-3]  |
|           | arg       |

# That's what we were seeing before

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
```

-fstack-protector-all

```
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

|                 |   |
|-----------------|---|
|                 | <pre>func(int, int, char*):<br/>    pushl    %ebp<br/>    movl     %esp, %ebp<br/>    subl     \$40, %esp</pre>   |
| copy arg1       | <pre>    movl     8(%ebp), %eax<br/>    movl     %eax, -28(%ebp)</pre>  |
| copy arg2       | <pre>    movl     12(%ebp), %eax<br/>    movl     %eax, -32(%ebp)</pre>   |
| copy arg3       | <pre>    movl     16(%ebp), %eax<br/>    movl     %eax, -36(%ebp)</pre>   |
| write<br>canary | <pre>    movl     %gs:20, %eax<br/>    movl     %eax, -12(%ebp)</pre>   |
|                 | <pre>    xorl     %eax, %eax<br/>    movl     \$-559038737, -20(%ebp)</pre>   |
|                 | <pre>    subl     \$8, %esp<br/>    pushl    -36(%ebp)<br/>    leal     -16(%ebp), %eax<br/>    pushl    %eax<br/>    call     strcpy<br/>    addl     \$16, %esp</pre> |
|                 | <pre>    nop<br/>    movl     -12(%ebp), %eax<br/>    xorl     %gs:20, %eax<br/>    je       .L4<br/>    call     __stack_chk_fail</pre>                                |
|                 | <pre>.L4:<br/>    leave<br/>    ret</pre>   |



# Additional Stack Canary Limitations

---

- Stack canaries do not protect from non-sequential overwrites.
  - Pointer subterfuge (mentioned earlier) or array index input
  - Printf vuln that allows targeted writing
- Stack canaries do not prevent the overwrite. They only attempt to detect it once it happens.
  - How to recover?

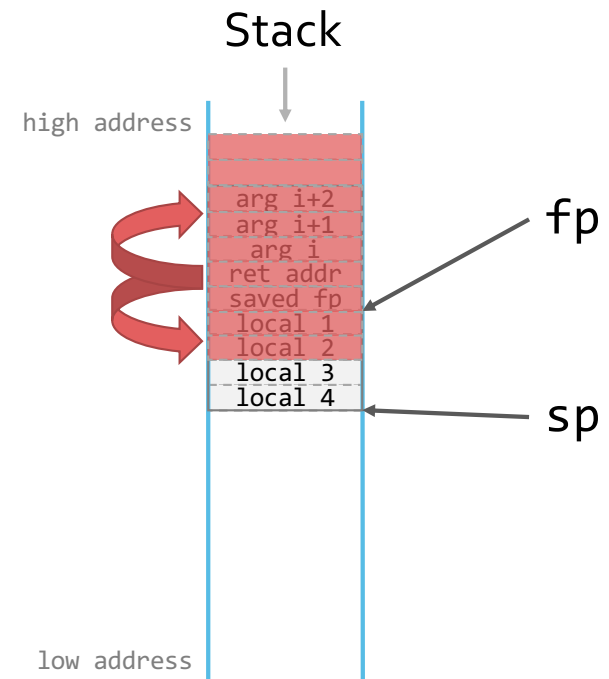
# Stack Canaries

---

- In spite of the above limitations, stack canaries still offer significant value for relatively little cost
- Considered essential mitigation on modern systems

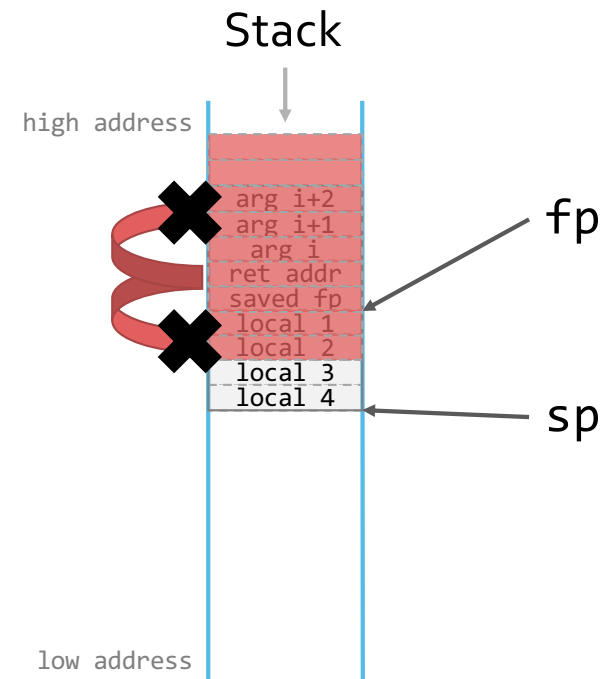
## Another idea: Nonexecutable Stack

- **Goal:** prevent execution of shellcode on stack
- Modern processors can mark virtual memory pages with permission bits: Read, Write, and/or eXecute access (RWX)



# Nonexecutable Stack

- Mark stack pages as “nonexecutable”
- Attempts to execute instructions from the stack trigger memory access violations



# Data Execution Prevention (DEP)

---

- Mitigation extends beyond the stack
- Make **all pages** either writeable or executable, but not both
  - Stack and heap are writeable, but not executable
  - Code is executable, but not writeable
  - Also known as W^X (Write XOR eXecute)

# DEP Tradeoffs

---

- **Pros:**

- No changes to application software
- Little/no performance impact

- **Cons:**

- Requires hardware support (MPU, MMU, or SMMU)
  - May not be doable on microcontrollers or some embedded processors
- Doesn't work with certain programming "tricks"
  - E.g., self-modifying code, compile in place

# DEP Limitations

---

- Assumptions:
  - If we prevent attackers from injecting code, we deny them ability to execute malicious code
  - All pages are either writeable or executable, but not both
- We won!  
... right?

# DEP Bypass

---

- What if some pages need to be both writeable and executable?
  - Special handling needed for JIT code, memory overlays, and self-modifying code
  - Is this common?
- What if there is useful executable code you can repurpose?
  - For example, transfer control flow to an existing function (e.g., libc's `exec()` or `system()` functions) that can be coerced into doing something bad
- Is there a way for an attacker to execute **arbitrary code** even without the ability to inject it into the victim process?
  - Next time...
  - Teaser: Yes, yes there is.



## Another idea: Hide the shell code's location

---

- Idea: If the attacker doesn't know where in memory their shellcode is stored, then its hard to make the processor jump there...

# Process Memory Map

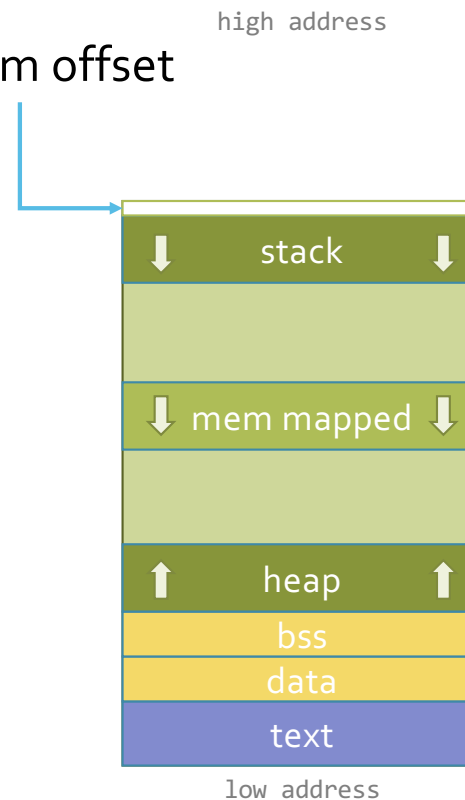
- **Assumption:** Stack smashing exploits depend on being able to predict stack addresses.



# Randomize Stack Base

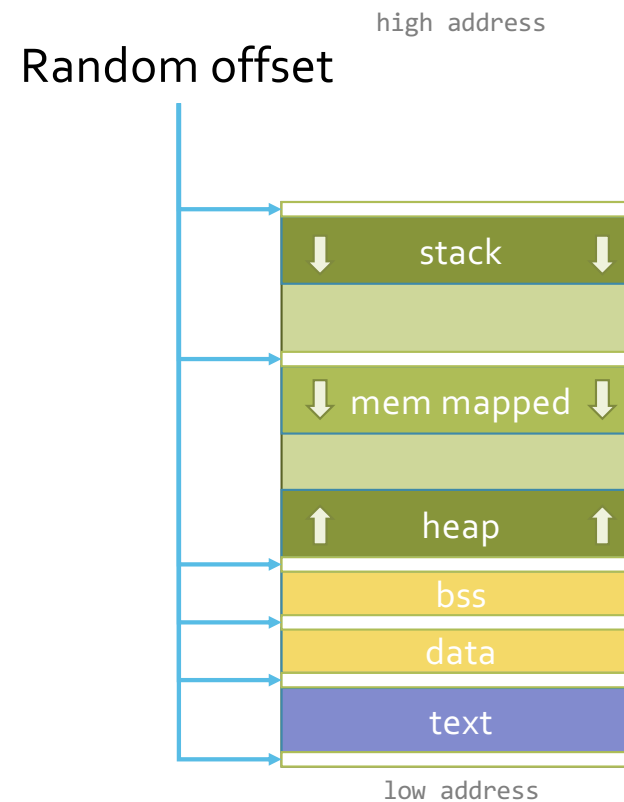
- Add a random offset to stack base.
- Assumption: harder for attackers to guess the location of their shellcode on the stack.
- Bypass?
  - Guess stack location
    - Similar to canary approach
  - Information leak
  - Longer NOP sled
  - Put shellcode on heap

Random offset



# Address Space Layout Randomization

- ASLR extends the concept to other sections of process memory.



<https://hyperboleandahalf.blogspot.com/2010/06/this-is-why-ill-never-be-adult.html>

# ASLR Tradeoffs

---

- Requires compiler, linker, and loader support
- Side Effects
  - Increases code size and performance overhead
  - Random number generator dependency (like canaries)
  - Potential load time impact for relocation
- Bypass
  - Information leaks/guessing
    - Servers that fork(), limitations in layout options
  - Heap Spraying

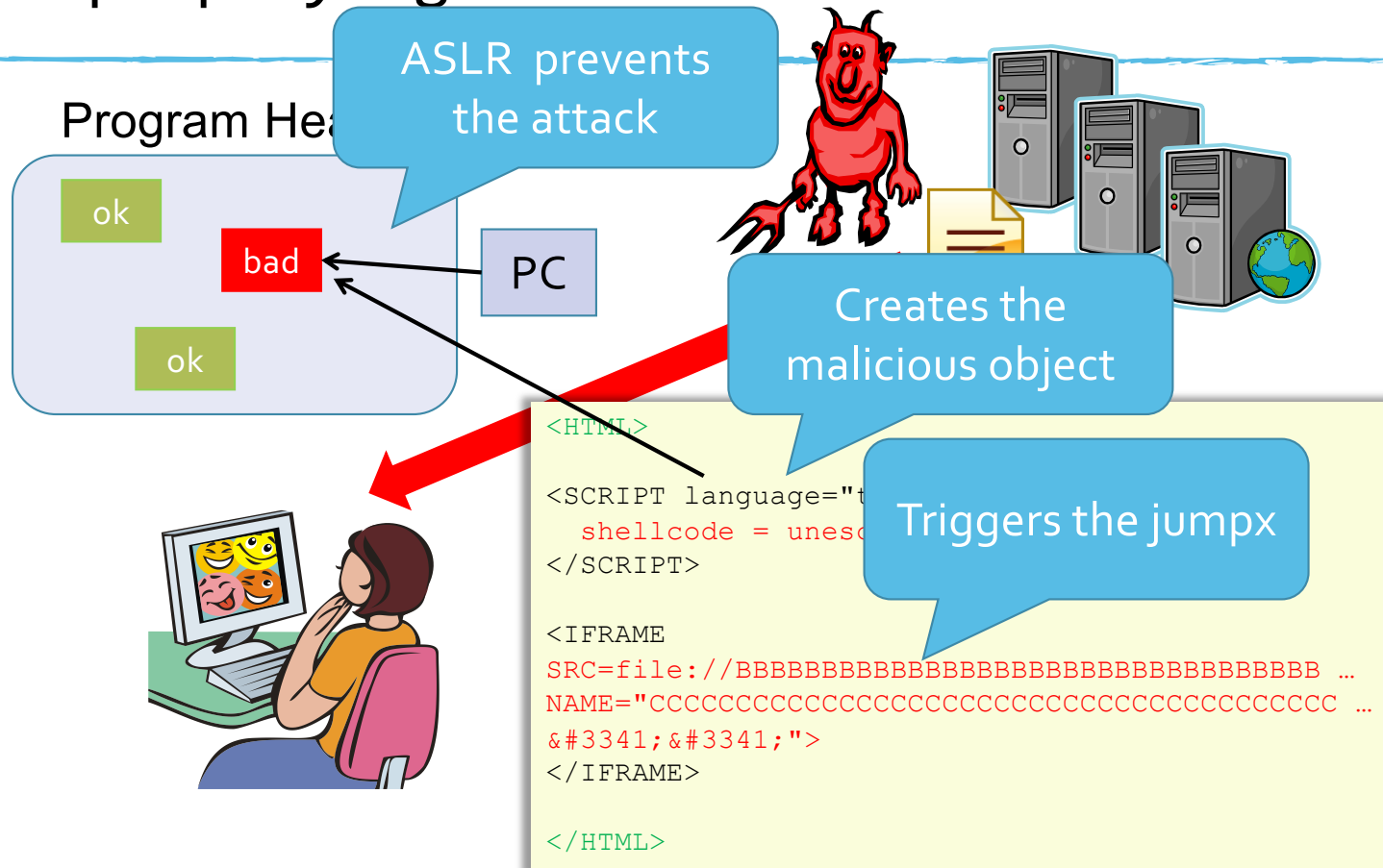
## Advanced techniques:

### Heap spray

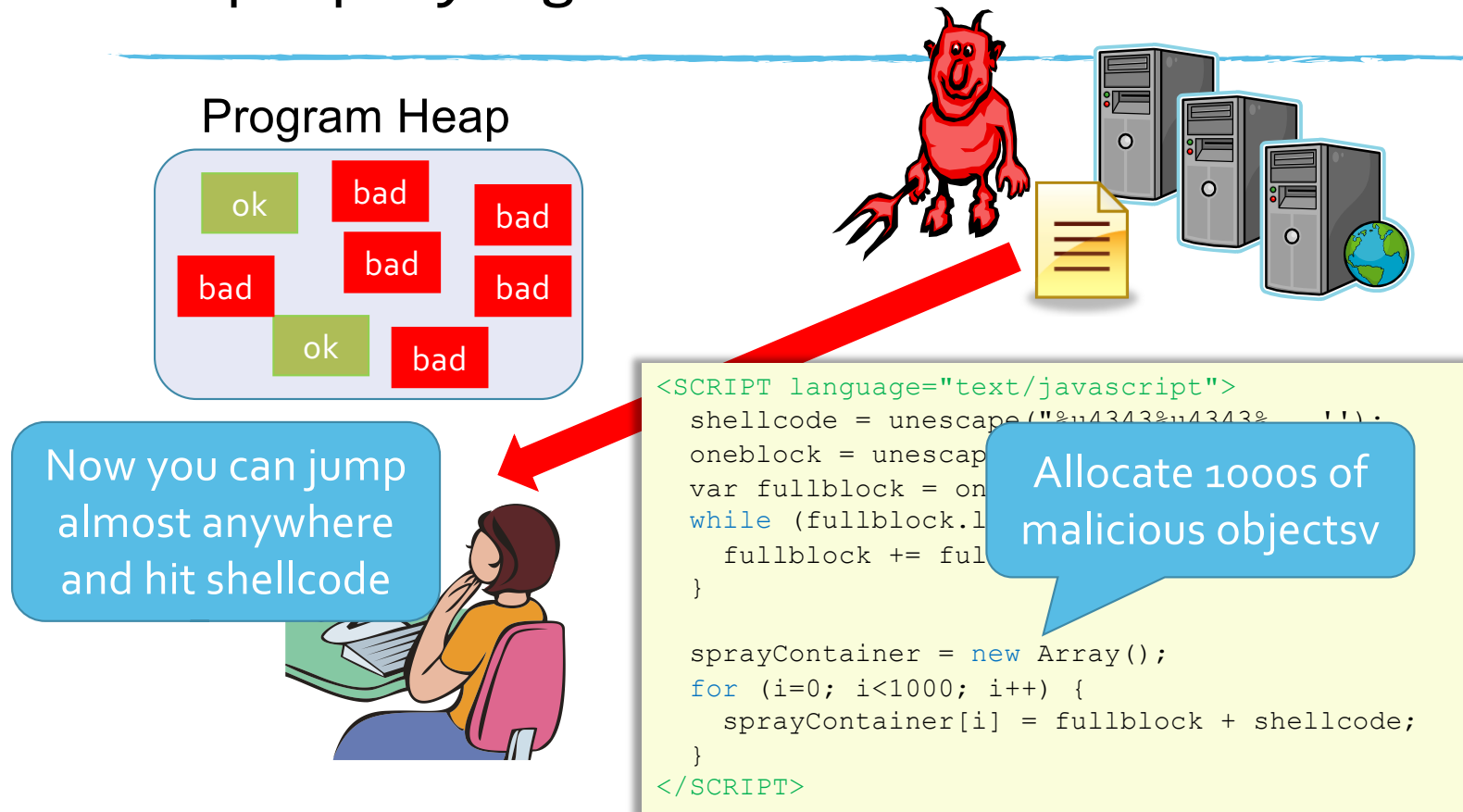
---

- Overflow can be used to cause control transfer into heap, but we don't know where shellcode is stored due to ASLR
- Basic bypass idea: use overwhelming force
  - Allocate jizillions of copies of the shellcode (with big NOP sleds) and then jump **blindly** into the heap
- Very common with “drive-by download” attacks on browsers
  - Heap spray implemented using Javascript

# Heap Spraying



# Heap Spraying





# Summary

---

- Mitigations
  - **Stack canaries/cookies**: detect overwrite of stack into control data
  - **DEP/W^X**: mark stack and heap as non-executable (hardware support) to prevent shell code from being located there
  - **ASLR**: randomize location of key data structures (e.g., stack and heap) to prevent attacks from knowing where they are
- Do not provide perfect security (all can be bypasses)
  - **Theory**: make reliable exploits more expensive to implement
  - **Practice**: varying effectiveness, typically decreases with time as new bypasses are developed

# Next Lecture...

---

Hardcore attacks and mitigations:  
Return-oriented programming and control flow integrity