SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 479/980: Systems and Network Security

Spring 2020

Control-flow Hijacking Defences

Instructor: Khaled Diab

Control-flow Hijacking Attacks so far…

- Buffer overflow: modify the return address
- Format string vulnerability: various range of attacks
- Heap overflows
- …

The Mistake!

Mixing code and data

→ Eventually, an attacker can inject code

→ Source of other attacks…

Defenses Overview

- Fix bugs
  - Automated tools
  - Rewrite software in different languages (examples?)
    - Legacy code?
- Run-time defenses:
  - StackGuard, Shadow Stack
- Platform defenses:
  - NOEXEC, ASLR

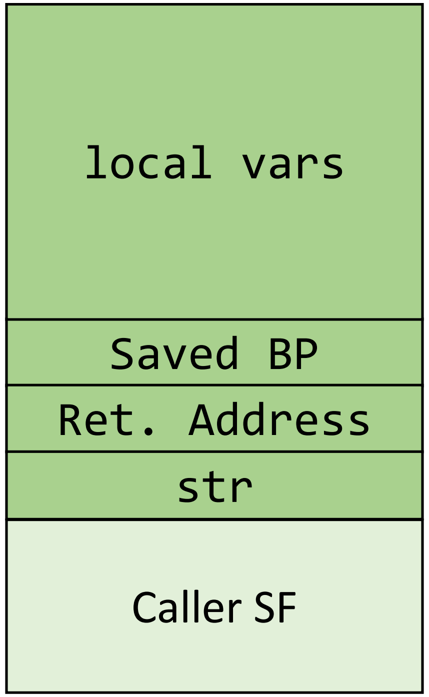
StackGuard

- 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)

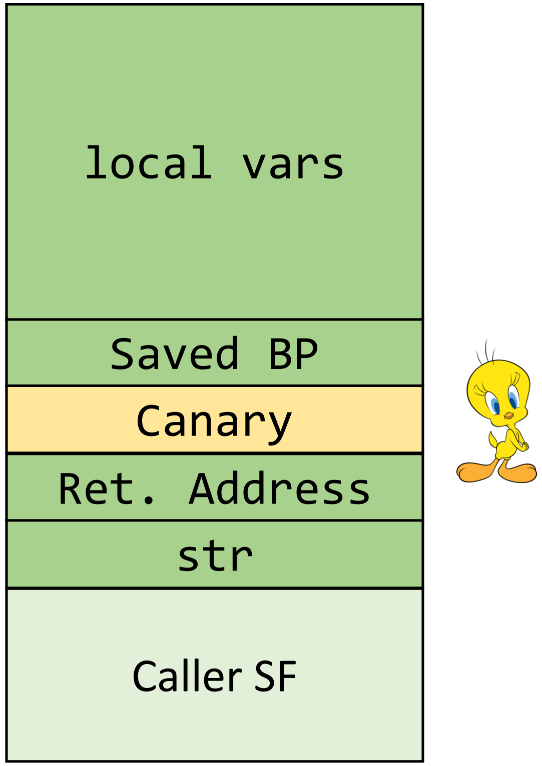
- Epilogue
  - pop the canary value from the stack and check that it hasn't changed

Stack (no canary)

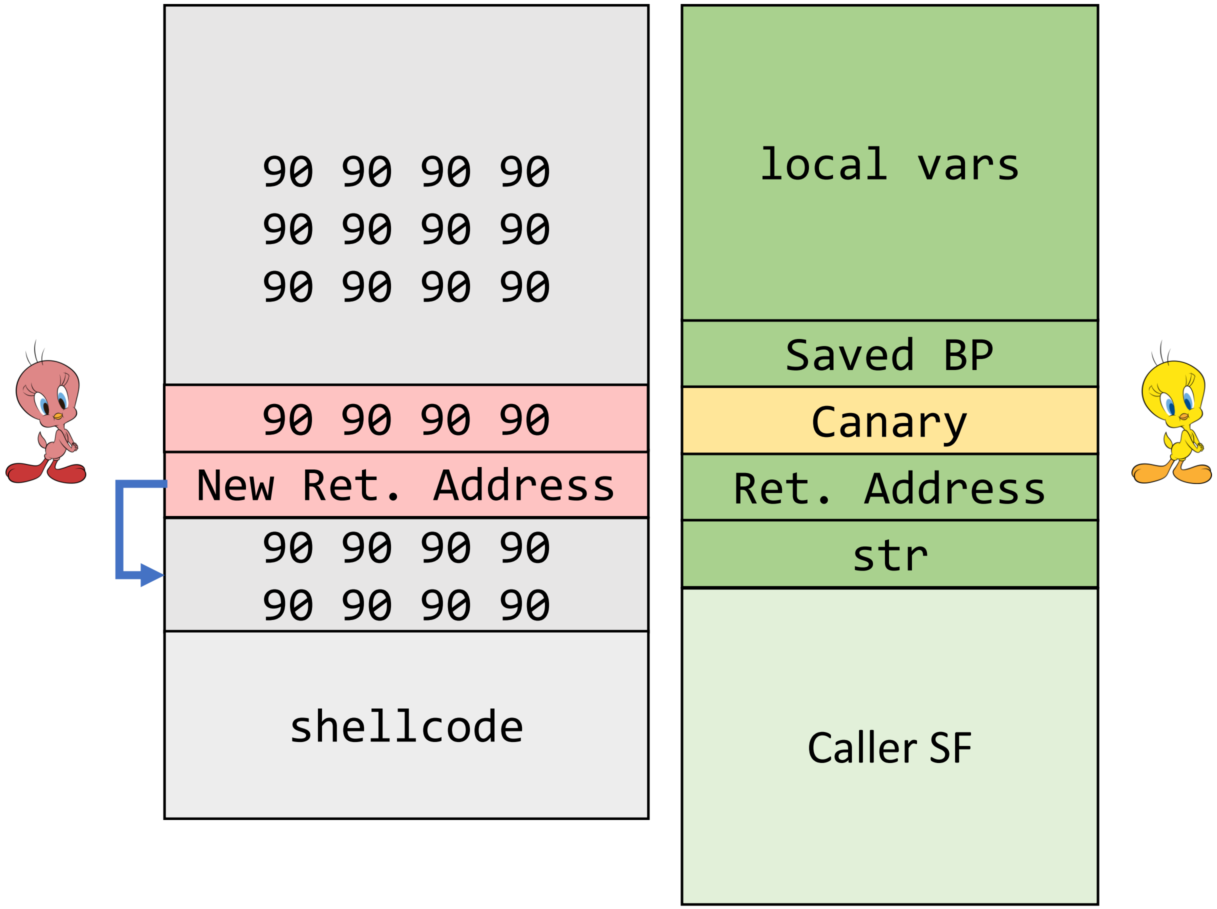


Stack + Canary

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
  - To act as string terminator for most string functions

Terminator Canary

00	0A	FF	0D

- 0D → Carriage Return
- FF → Form feed
- 0A → LF: Terminates `gets`
- 00 → Null: Terminates `strcpy`

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

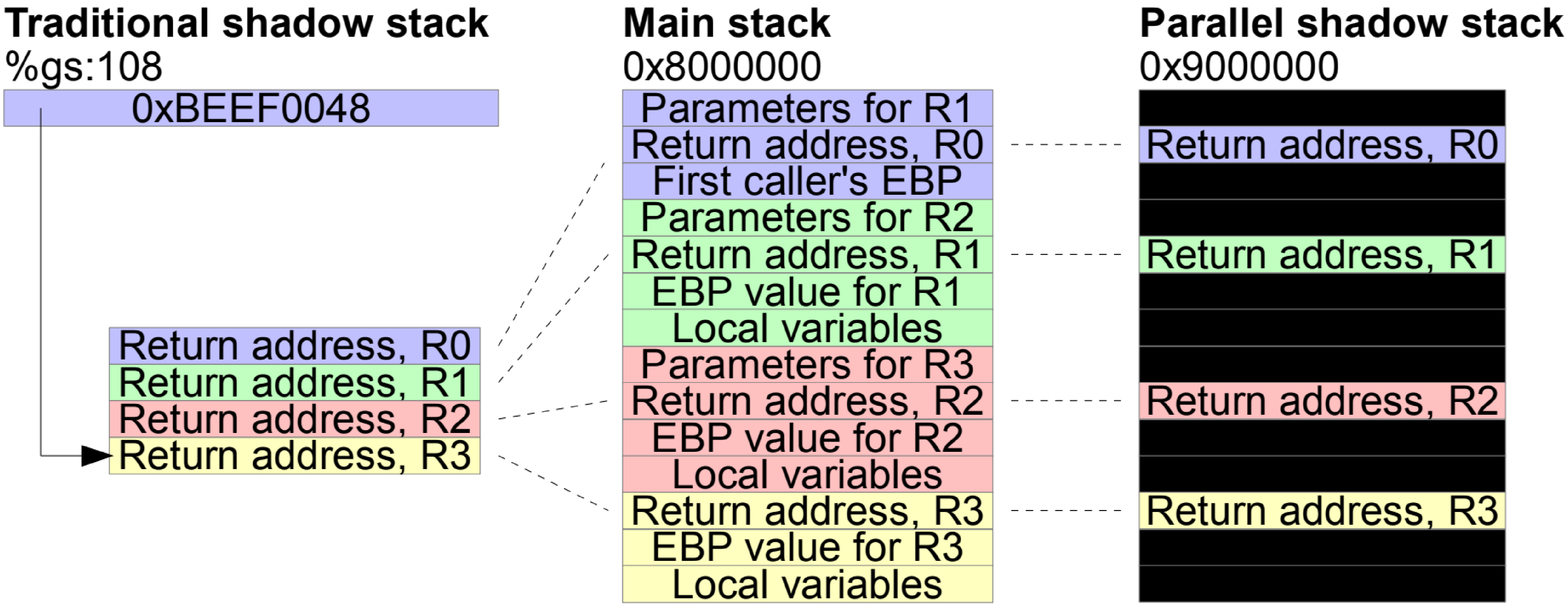
- Chrome OS team has another proposal
  - `-fstack-protector-strong`
    - A superset of `-fstack-protector`
    - Examples: if a function has an array
    - More details…

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

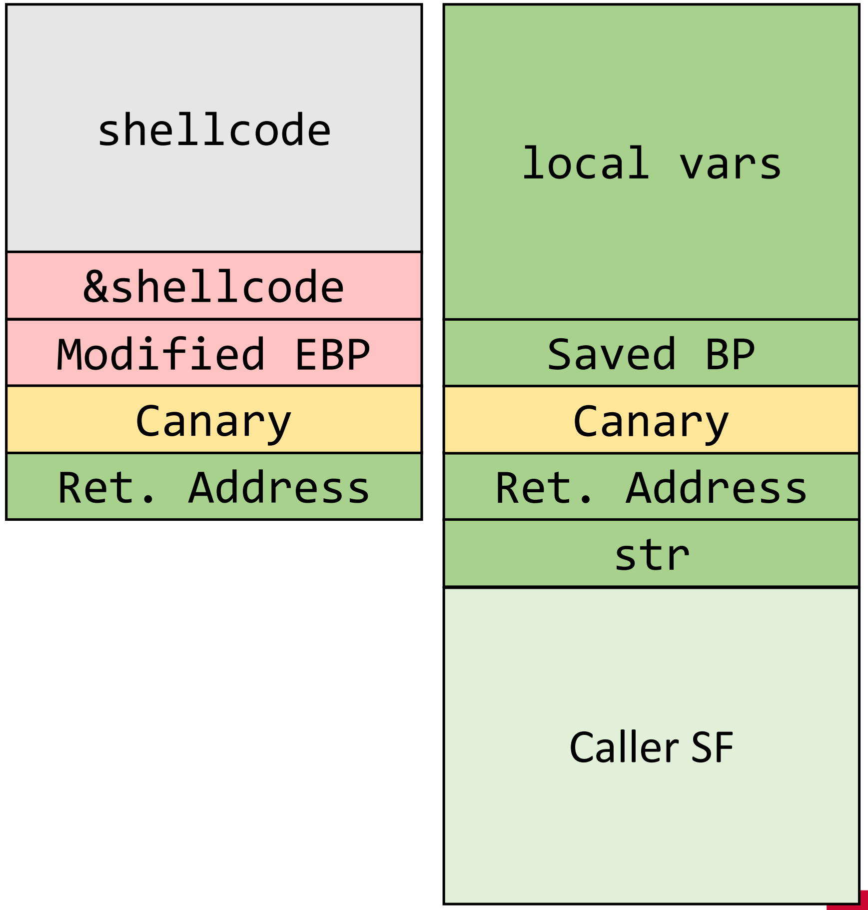
What is the main assumption so far?

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!



NOEXEC

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

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

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

- 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);
```

How can we address these issues?

- Project ideas?

Next lecture…

- return-to-libc
- ROP
- Control flow integrity