memory safety

non-executable pages, stack canary, PACs, ASLR

slides
bit.ly/cs161-disc

feedback
bit.ly/extended-feedback

general questions, concerns, etc.

hack(s) of the day

hack(s) of the day

- An Atlassian product is vulnerable (again)

hack(s) of the day

- An Atlassian product is vulnerable (again)
- attackers can gain access to server management instance by intercepting tokens

hack(s) of the day

- An Atlassian product is vulnerable (again)
- attackers can gain access to server management instance by intercepting tokens
- similar attack vector to command injection flaw with BitBucket

hack(s) of the day

- An Atlassian product is vulnerable (again)
- attackers can gain access to server management instance by intercepting tokens
- similar attack vector to command injection flaw with BitBucket
- ImageMagick - image processing web package
    - DoS if image filename is “-”, can also embed remote file info based on PNG content

vulnerability recap?

stack smashing,
signed/unsigned, etc.

[skip](#)

recap: overwriting the rip

- we have 28 bytes of shellcode
- place shellcode after rip
- `'A' * 24 + '\x5c\xcd\xff\xbf' + SHELLCODE`


```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

mitigating the `gets` vulnerability

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

```
void safe(void) {
    char name[20];
    ...
    fgets(name, 20, stdin);
    ...
}
```

specify length!

signed/unsigned vulnerabilities

signed

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

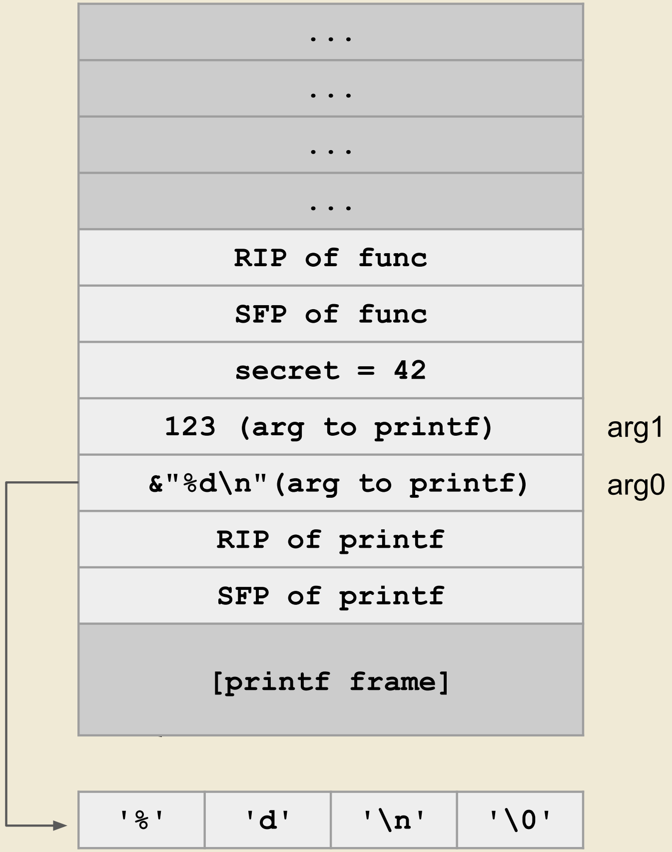
```
void *memcpy(void *dest, const void *src, size_t n);
```

unsigned

printf vulnerability

```
void func(void) {
    int secret = 42;
    printf("%d\n", 123);
}
```

two arguments

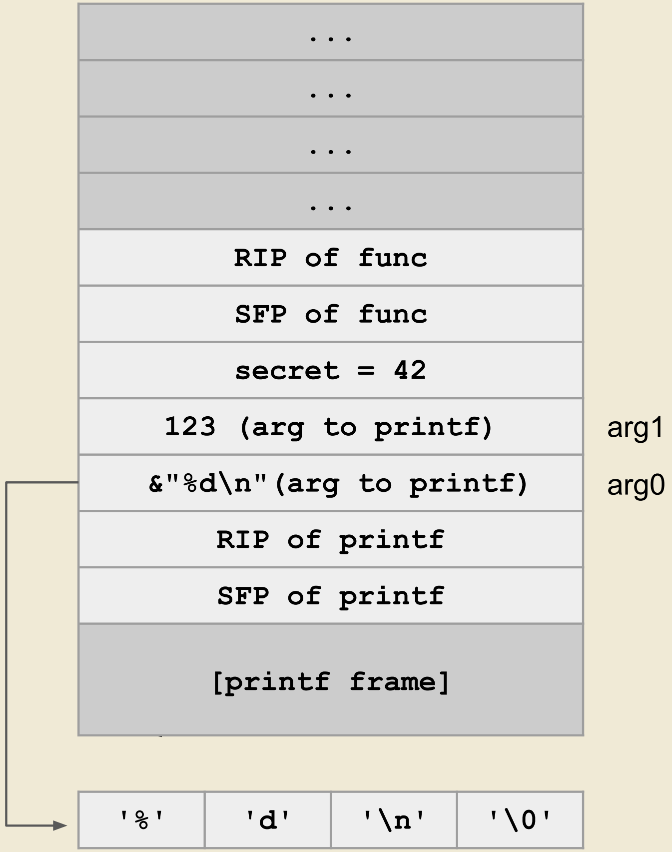


printf vulnerability

```
void func(void) {
    int secret = 42;
    printf("%d\n", 123);
}
```

two arguments

what if there’s only one?



printf vulnerability

```
void func(void) {
    int secret = 42;
    printf("%d\n");
}
```

one argument

what if there's only one?

...	
...	
...	
...	
RIP of func	
SFP of func	
secret = 42	arg1
&"%d\n"(arg to printf)	arg0
RIP of printf	
SFP of printf	
[printf frame]	

printf attack common parameters

parameters	output
%p	representation as a pointer to void
%d	decimal
%c	character
%u	unsigned decimal
%x	hexadecimal
%s	string
%n	write number of characters printed into a pointer

mitigating printf vulnerabilities

```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

```
void vulnerable(void) {
    char buf[64];
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf("%s", buf);
}
```

only accept trusted input!

memory safety defenses

non-executable pages, stack canary, PACs, ASLR

non-executable pages

non-executable pages

- typically, stack, heap, and static data **writable but not executable**

non-executable pages

- typically, stack, heap, and static data **writable but not executable**
- code is **executable but not writable**

non-executable pages

- typically, stack, heap, and static data **writable but not executable**
- code is **executable but not writable**
- why?

non-executable pages

- typically, stack, heap, and static data **writable but not executable**
- code is **executable but not writable**
- why?
    - do we often need to execute code within the stack?

non-executable pages

- typically, stack, heap, and static data **writable but not executable**
- code is **executable but not writable**
- why?
  - do we often need to execute code within the stack?
  - prevents executing shellcode within stack

non-executable pages

- typically, stack, heap, and static data **writable but not executable**
- code is **executable but not writable**
- why?
    - do we often need to execute code within the stack?
    - prevents executing shellcode within stack
- problem: <u>existing code</u> can still be used

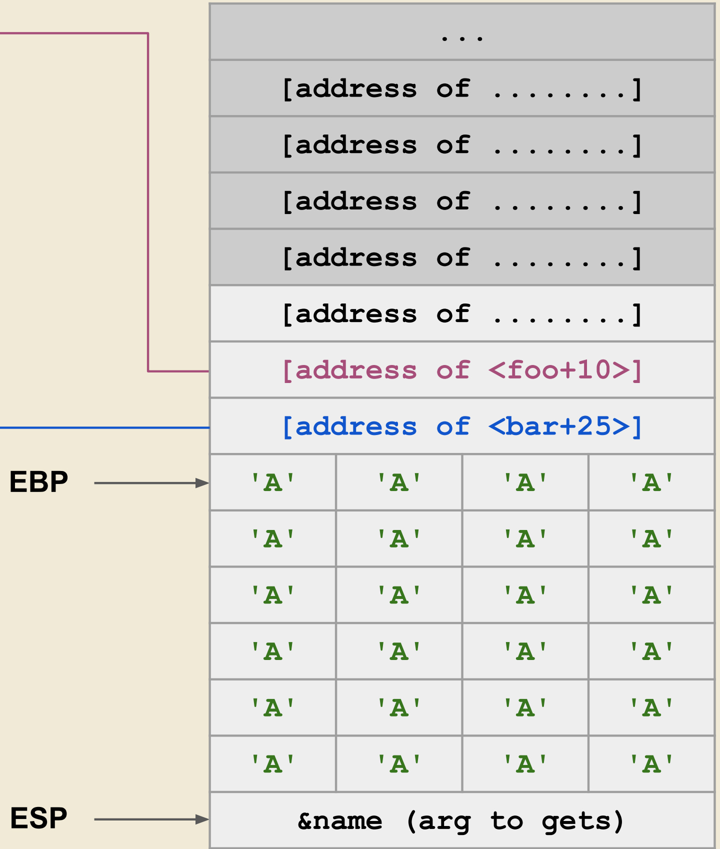
return-oriented programming

Exploit:

```
'A' * 24
    + [address of <bar+25>]
    + [address of <foo+10>]
    + ... (more chains)
```

```c
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
foo:
        addl $4, %esp
        xorl %eax, %ebx
        ret

bar:
        ...
        andl $1, %edx
        movl $1, %eax
        ret

vulnerable:
        ...
        call gets
EIP →   addl $4, %esp
        movl %ebp, %esp
        popl %ebp
        ret

main:
        ...
        call vulnerable
        ...
```



stack canary

stack canary

- how can we check that the RIP hasn't been overwritten?

stack canary

- how can we check that the RIP hasn’t been overwritten?
- idea—a way to check if the area around the RIP has been tampered with
  - like a canary in a coal mine!

stack canary

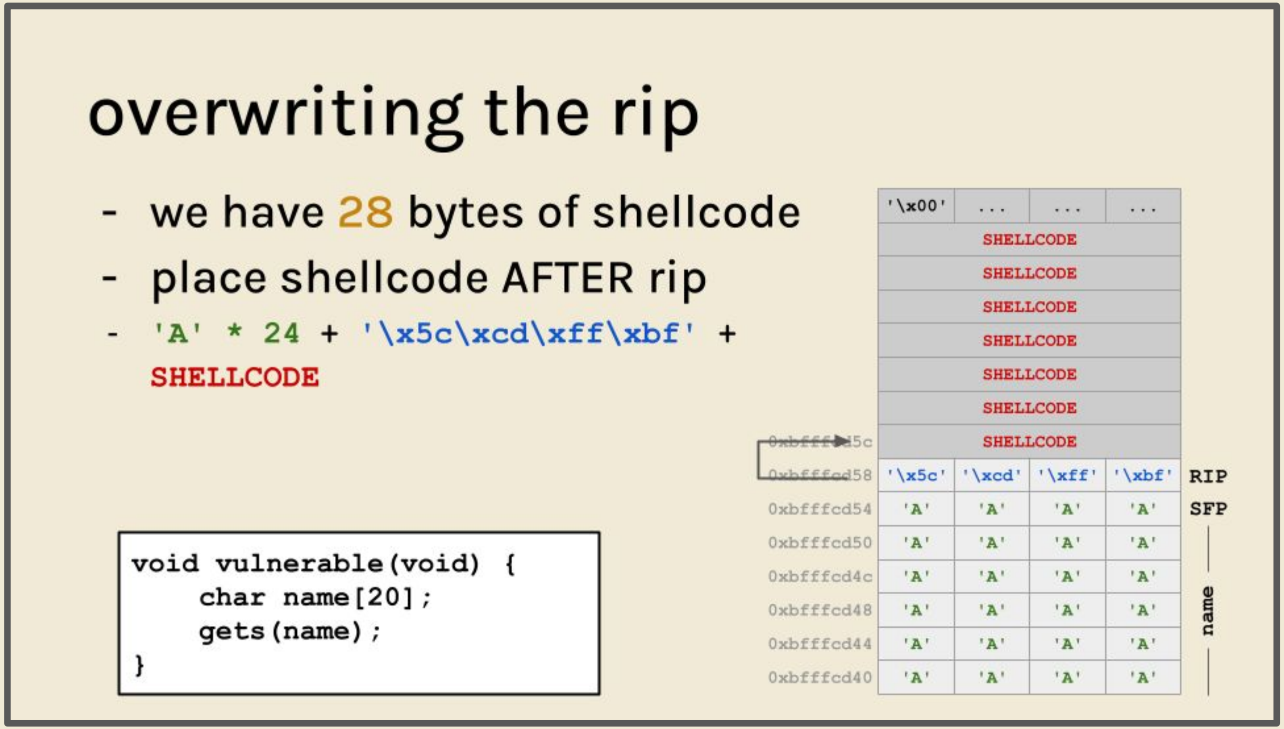
- how can we check that the RIP hasn’t been overwritten?
- idea—a way to check if the area around the RIP has been tampered with
  - like a canary in a coal mine!
- generate a random number, check that it’s the same after execution

stack canary

- consider our “overwriting the RIP” strategy

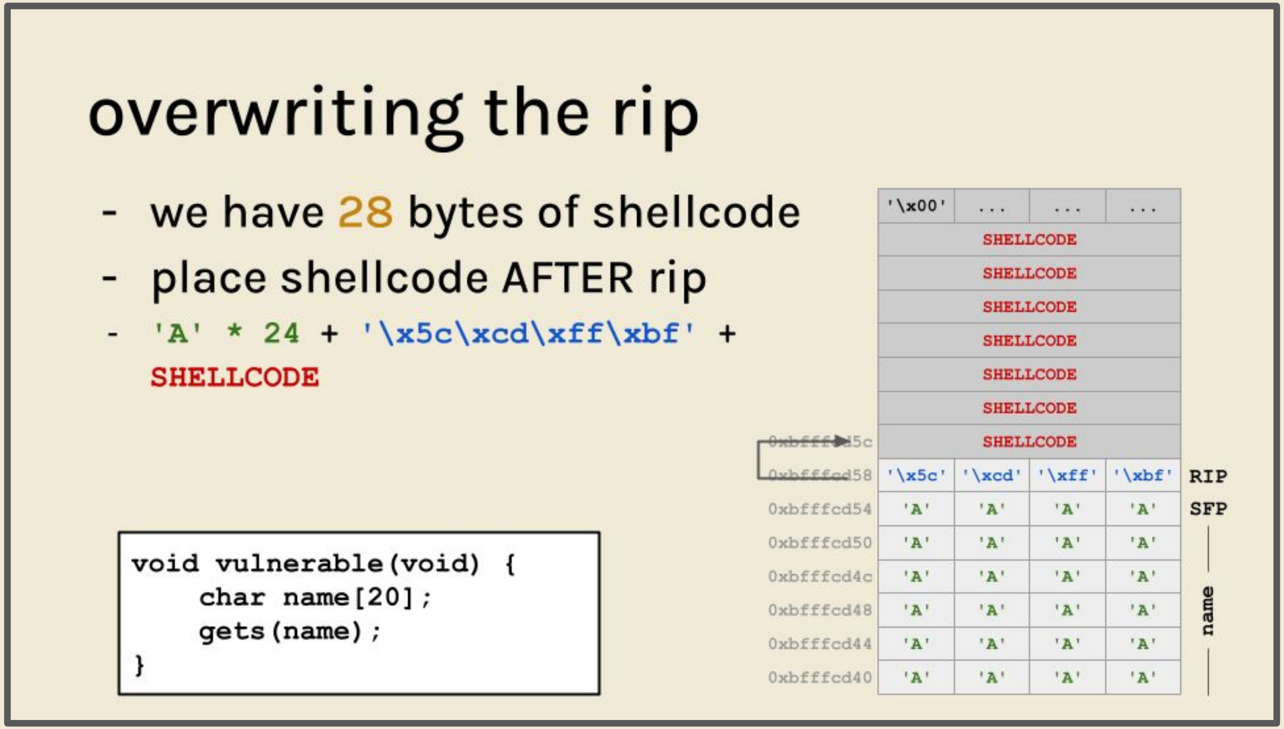
stack canary

- consider our “overwriting the RIP” strategy



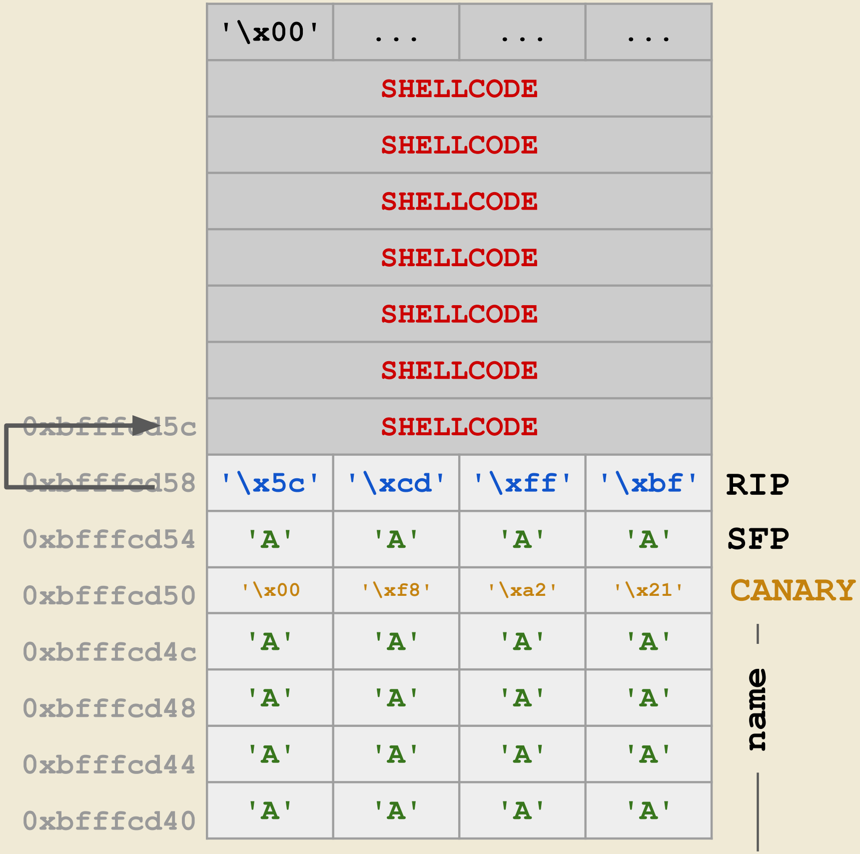
stack canary

- consider our “overwriting the RIP” strategy
- how can we stop this from happening?



stack canary

- our canary (0x21a2f900) is overwritten with 'A' s

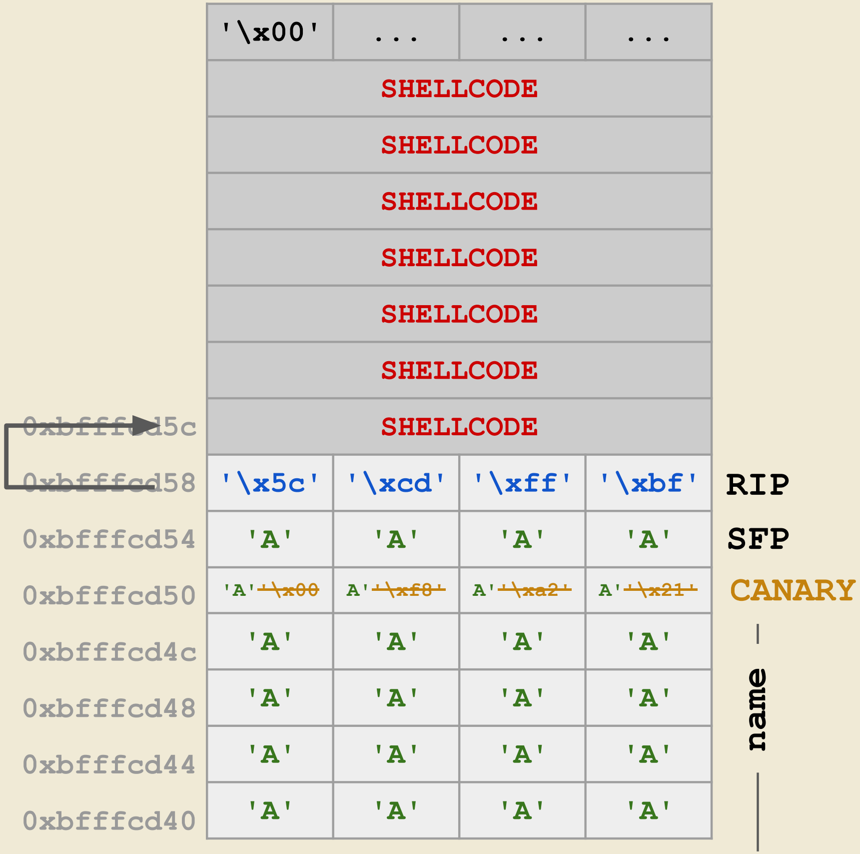



```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

stack canary

- our canary (0x21a2f900) is overwritten with **'A'** s

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



stack canary: properties

stack canary: properties

- randomly generated each time a program runs, same for all functions within a run

stack canary: properties

- randomly generated each time a program runs, same for all functions within a run
- first byte is NULL to prevent string-based attacks

stack canary: properties

- randomly generated each time a program runs, same for all functions within a run
- first byte is NULL to prevent string-based attacks
  - challenge: what might this attack look like?

subverting the canary

subverting the canary

- format string vulnerabilities that allow writing to arbitrary memory locations

subverting the canary

- format string vulnerabilities that allow writing to arbitrary memory locations
- heap overflows write to heap, never mess with <u>stack</u> canary

subverting the canary

- format string vulnerabilities that allow writing to arbitrary memory locations
- heap overflows write to heap, never mess with <u>stack</u> canary
- vtable exploits (C++)

subverting the canary

- format string vulnerabilities that allow writing to arbitrary memory locations
- heap overflows write to heap, never mess with <u>stack</u> canary
- vtable exploits (C++)
- guess the canary value (64 bits - 8 bits (NULL))

pointer authentication

pointer authentication

- unused bits in a 64 bit address are used to store PACs (pointer authentication codes)

pointer authentication

- unused bits in a 64 bit address are used to store PACs (pointer authentication codes)
- deterministic sequence of numbers (like canary) associated with EACH address

pointer authentication

- unused bits in a 64 bit address are used to store PACs (pointer authentication codes)
- deterministic sequence of numbers (like canary) associated with EACH address
  - each address has its own PAC

pointer authentication

- unused bits in a 64 bit address are used to store PACs (pointer authentication codes)
- deterministic sequence of numbers (like canary) associated with EACH address
    - each address has its own PAC
    - PAC generated from CPU master secret

pointer authentication

- unused bits in a 64 bit address are used to store PACs (pointer authentication codes)
- deterministic sequence of numbers (like canary) associated with EACH address
  - each address has its own PAC
  - PAC generated from CPU master secret
    - can’t be observed by program

pointer authentication

0xf8a112c4

pointer authentication

0xf8a112c4

pointer authentication

$$0x\underbrace{f8a}_{\text{unused bits (PAC)}}112c4$$

pointer authentication

address (0x000112c5)

0xf8a112c4

unused bits (PAC)

subverting PACs

subverting PACs

- guess PAC (typically 44 used bits, 20 unused)

subverting PACs

- guess PAC (typically 44 used bits, 20 unused)
- learn the CPU master secret (OS vulnerability)

subverting PACs

- guess PAC (typically 44 used bits, 20 unused)
- learn the CPU master secret (OS vulnerability)
- pointer reuse

ASLR (address space layout randomization)

ASLR (address space layout randomization)

- put each segment of memory in a different location each time the program is run

ASLR (address space layout randomization)

- put each segment of memory in a different location each time the program is run
- relative locations are the same, just the absolute addresses of the start of the heap, stack, code are randomized

subverting ASLR

subverting ASLR

- leak a pointer (RIP, a stack pointer, etc.) and use **relative addressing** to figure out address

subverting ASLR

- leak a pointer (RIP, a stack pointer, etc.) and use **relative addressing** to figure out address
  - variables , code, etc. still the same **relative** distance from each other

worksheet
(on 161 website)

feedback

bit.ly/extended-feedback

slides: **bit.ly/cs161-disc**