Software Security: Exploitation

Shuai Wang

香 港 科 技 大 學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Software bugs that can be potentially exploited

- The so-called "vulnerability"

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Stack overflow
- Heap overflow
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

But note that many of the (logic) bugs are not exploitable and therefore are not "vulnerabilities"

Common Software Vulnerabilities

- Buffer overflows ← this time
- Format string problems
- Integer overflows
- Use after free ← next time
- Type confusion
- Race conditions…

Attacks marked in red are the most popular ones.

Buffer Overflow

- Used in 1988's Morris Internet Worm
- Alphe One's "*Smashing The Stack For Fun And Profit"* in 1996 popularizes stack buffer overflows
  - Your reading article today..
- Still extremely common today
  - Over 50% of advisories published by CERT (computer security incident report team) are caused by various buffer overflows

Computer Memory

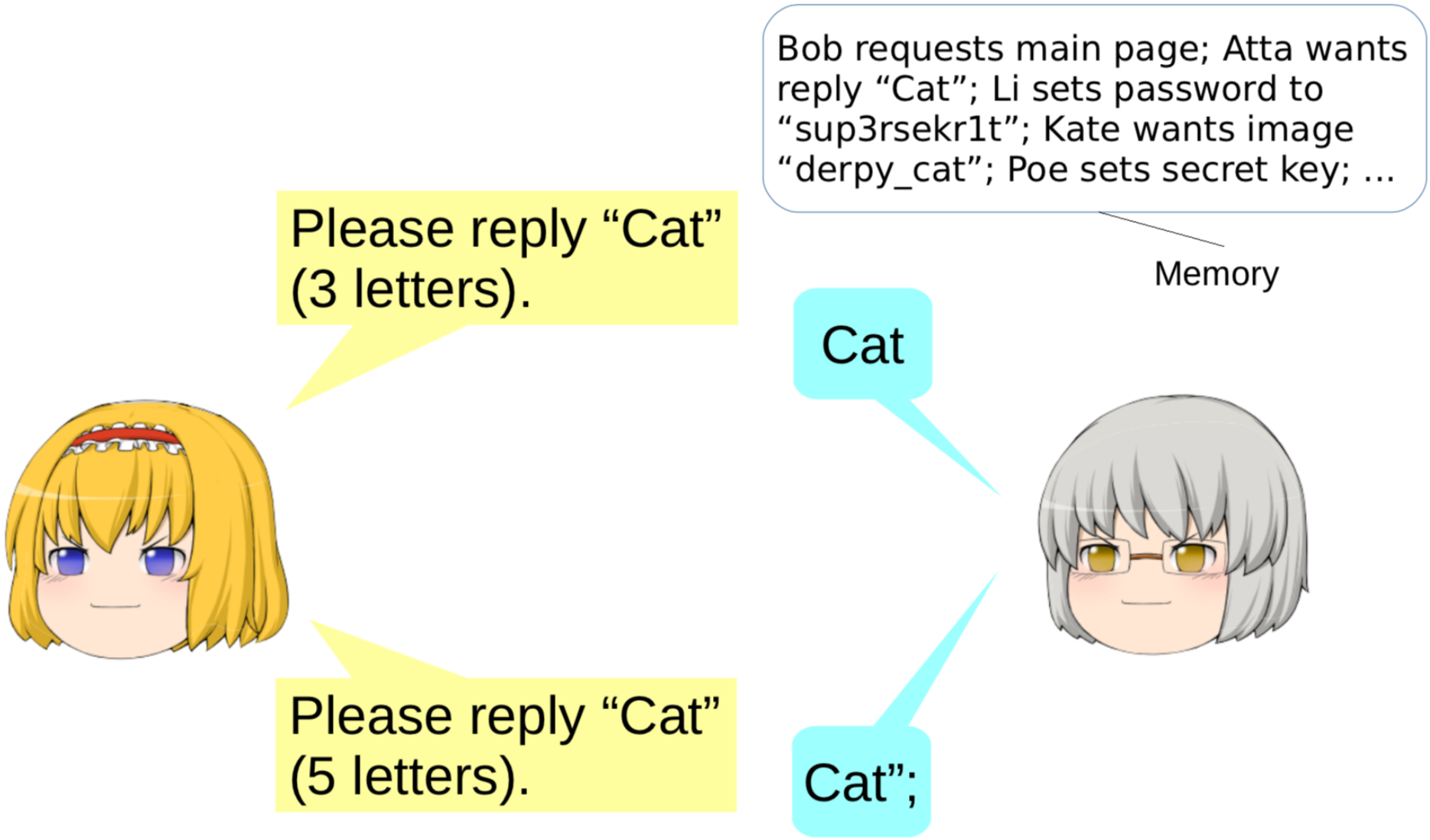
Your own memory may look like this:

> wake up; have breakfast; need to buy milk; turn off the lights; go to class; that man has a strange shirt; fall asleep; wake up

A web server's memory may look like this:

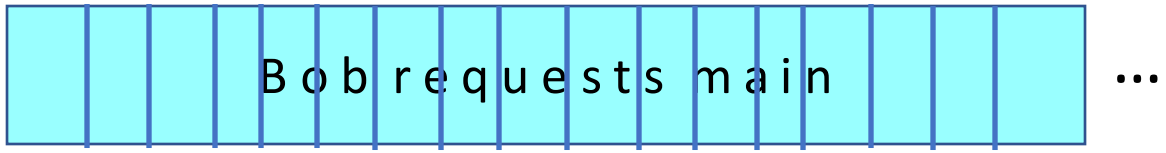
> Bob requests main page; Atta wants reply "Cat"; Li sets password to "sup3rsekr1t"; Kate wants image "derpy_cat"; Poe sets secret key; ...

Memory Read


Bob requests main page; Atta wants reply “Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key; …
Memory
Please reply “Cat” (3 letters).
Cat
Please reply “Cat” (5 letters).
Cat”;

Under the hood

```
char buffer[100];
```



Buffer is a data storage area inside computer memory

- Intended to hold pre-defined amount of data
- If more data is stored/required from it, it tampers the adjacent memory

Memory/Buffer Overread

Bob requests main page; Atta wants reply “Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key; ...

Memory

Please reply “Cat” (100 letters).

Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key; ...

Exploiting Memory For Fun and Profit

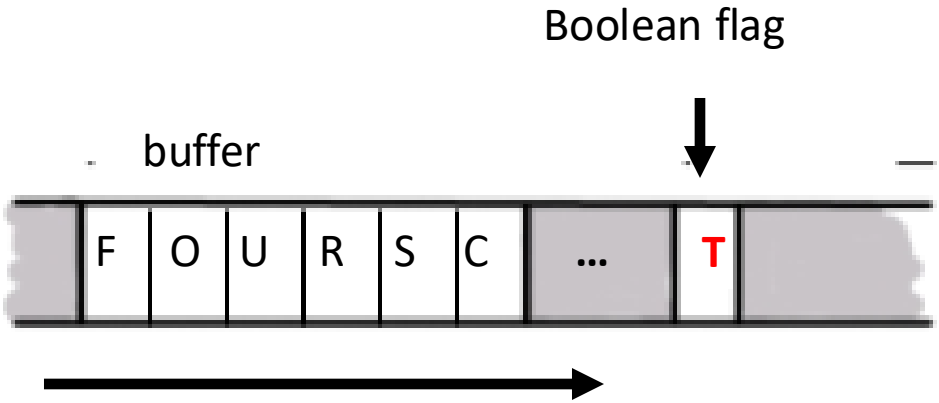
- All right, we talked about how buffer can be overread
  - The “Heartbleed” attack on OpenSSL
- Then, what about buffer overwritten?
  - Change the memory content?
  - What else?
    - Note that “code” can also be written in the memory… → executing arbitrary code!

Buffer Overwrite Scenario

- Users enter data into a Web form
- Web form is sent to server
- Server writes data to array called buffer, without checking length of input data
- Data "overflows" buffer
  - Such overflow might enable an attack
  - If so, attack could be carried out by anyone with Internet access

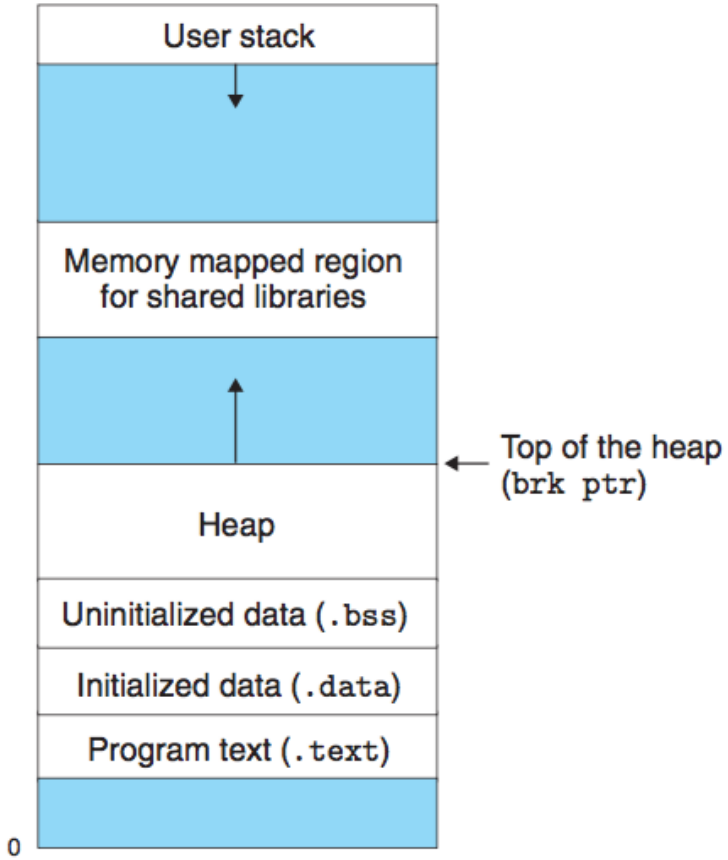
Simple Buffer Overflow

- Consider boolean flag to check password correct or not
- Buffer overflow could overwrite flag allowing anyone to authenticate

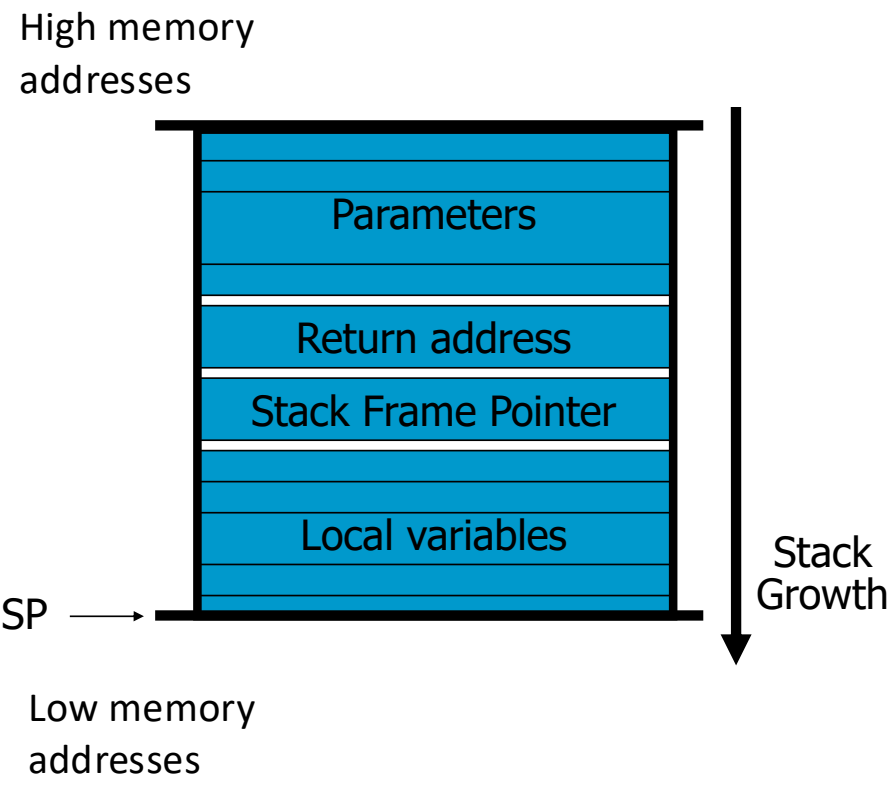


How is memory implemented?

- Linux standard memory layout for a process but Mac/Windows are very similar.



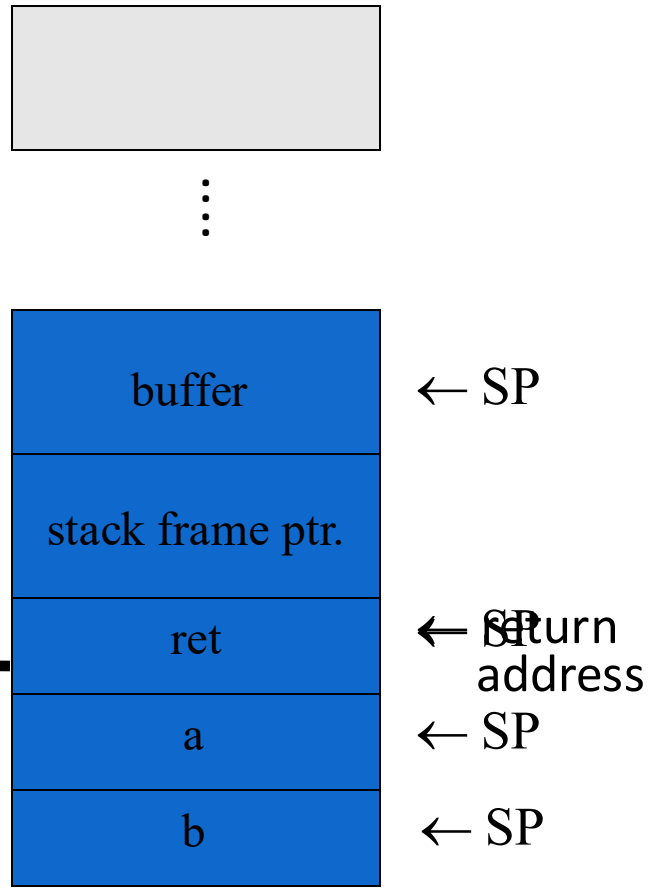
A Stack Frame



- A stack consists of logical stack frames that are pushed when calling a function and popped when returning.
- When a function is called, the return address, stack frame pointer and the variables are pushed on the stack.
- The stack grows from high address to low address.
  - When we overflow the buffer, the return address will be overwritten.

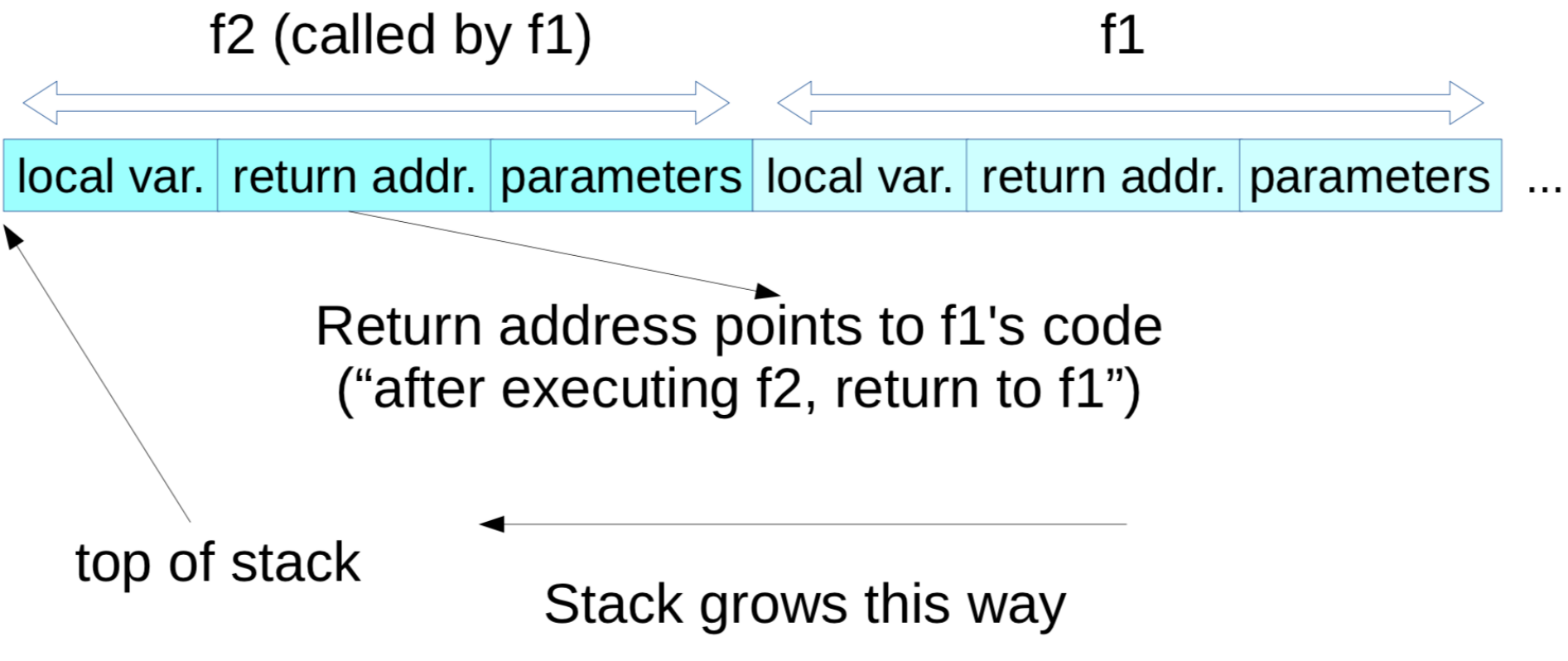
Stack Example

```
void func(int a, int b){
   char buffer[10];
   return;
}
void main(){
   func(1,2);
   int c = 1 + 1;
}
```



A Sequence of Stack Frames

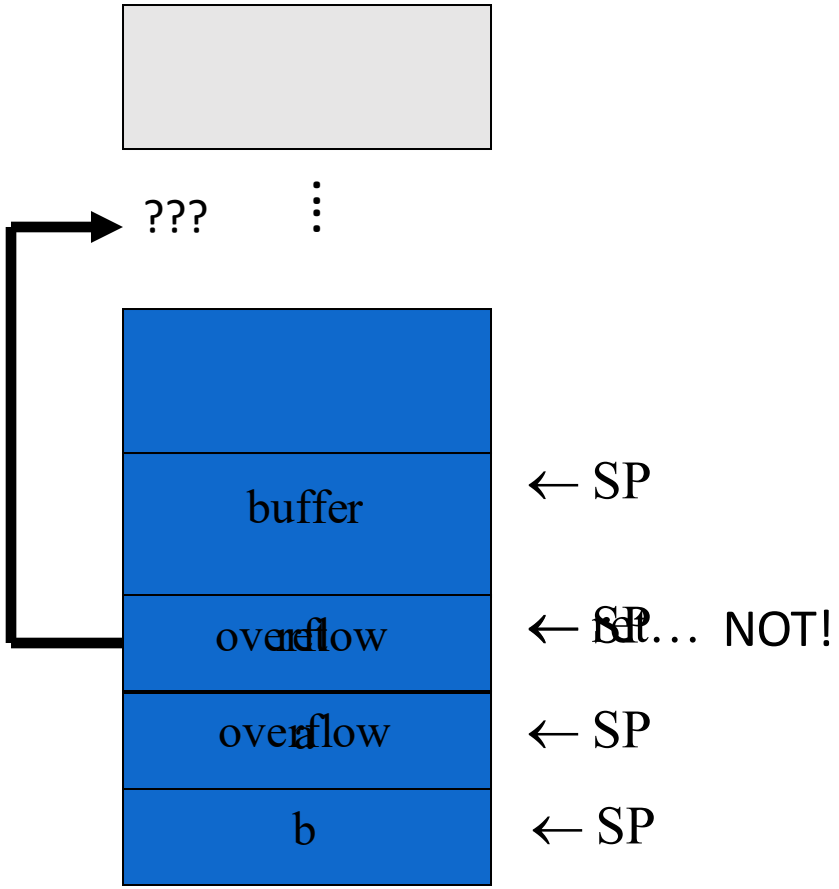
A simplified function stack (ignored stack frame pointer)!



Smashing the Stack

Ignored stack frame pointer since it does not affect our discussion!

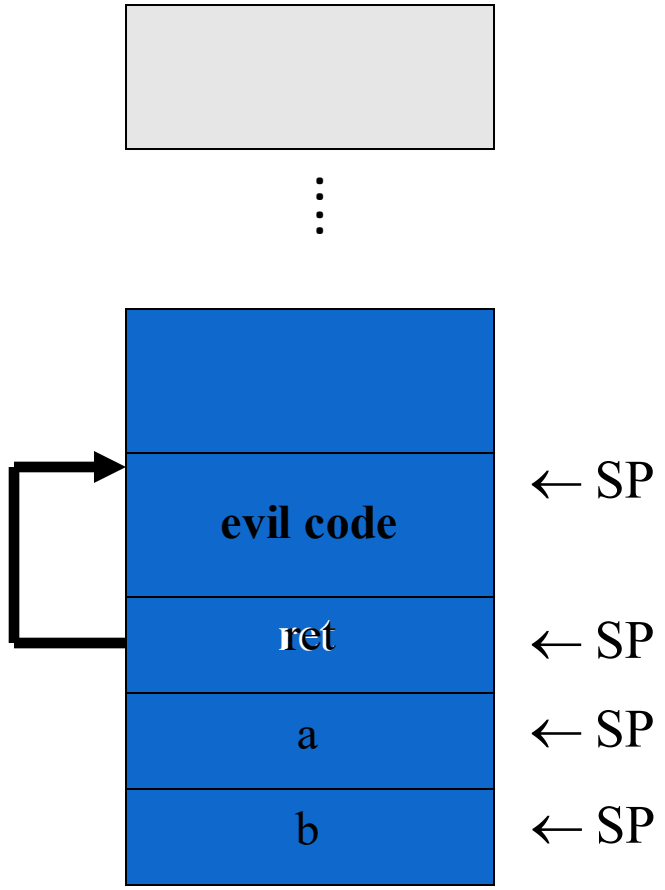
- What happens if buffer overflows?
- Program "returns" to wrong location
- A crash is likely



Smashing the Stack

Ignored stack frame pointer since it does not affect our discussion!

- An even better idea…
- **Code injection**
- Attacker can run arbitrary code on your machine
  - And even more ← later



Buffer Overflow Attack

- Also known as “stack smashing”, “buffer overrun”
  - Also known as a classic example of code injection attack, control-flow hijacking attack
- A buffer overflow must exist in the code
  - But not all buffer overflows are exploitable
- If exploitable, attacker can **inject code**
- Trial and error is likely required
  - Suppose we need to guess the address of evil code
  - But we only need one success anyway…
- Stack overflow is “attack of the decade”…
  - Also heap overflow and so on.

Buffer Overflow Attack

- Buffer grows backwardly on the stack and therefore overwrites the return address.
- Any thoughts on the mitigation…?
- What if buffer grows forwardly?
  - No, it does not solve the problem.
  - See my notes.

Unsafe C/C++ Lib Functions

strcpy (char *dest, const char *src)

gets (char *s)

strcat (char *dest, const char *src)

scanf (const char *format, …)

sprintf (conts char *format, …)

…

Buffer Overflow Attack due to Unsafe C/C++ Library Functions

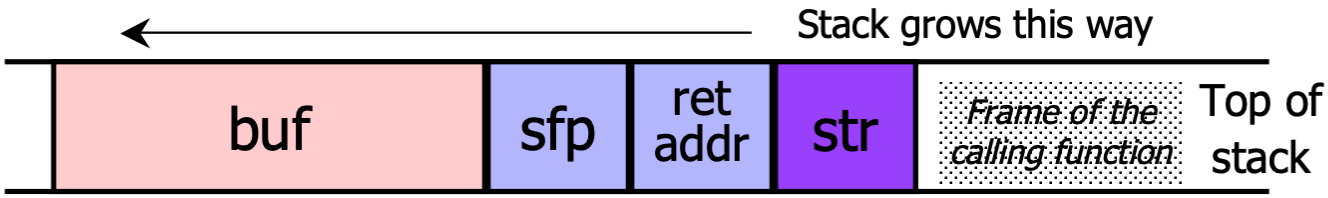
- Suppose Web server contains this function

```
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

Allocate local buffer (126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame with local variables is pushed onto the stack



https://www.cs.usfca.edu/~ejung/courses/686/lectures/19buffer.pdf

Buffer Overflow Attack due to Unsafe C/C++ Library Functions

- Memory pointed to by str is copied onto stack…

```
void func(char *str) {
     char buf[126];
     strcpy(buf,str);
}
```

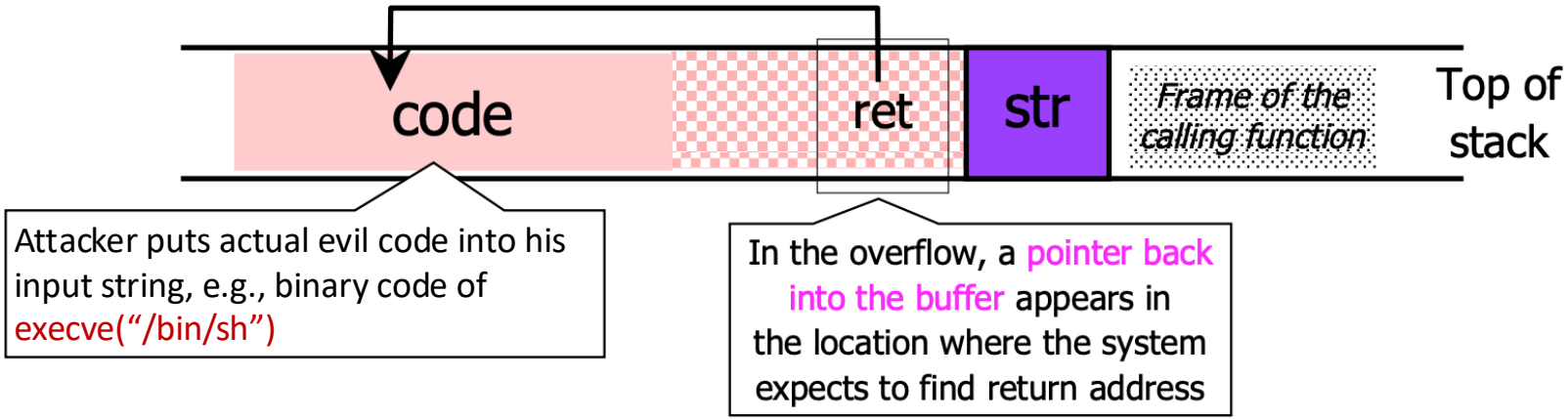
strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations

https://www.cs.usfca.edu/~ejung/courses/686/lectures/19buffer.pdf

Buffer Overflow Attack due to Unsafe C/C++ Library Functions

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon



- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

https://www.cs.usfca.edu/~ejung/courses/686/lectures/19buffer.pdf

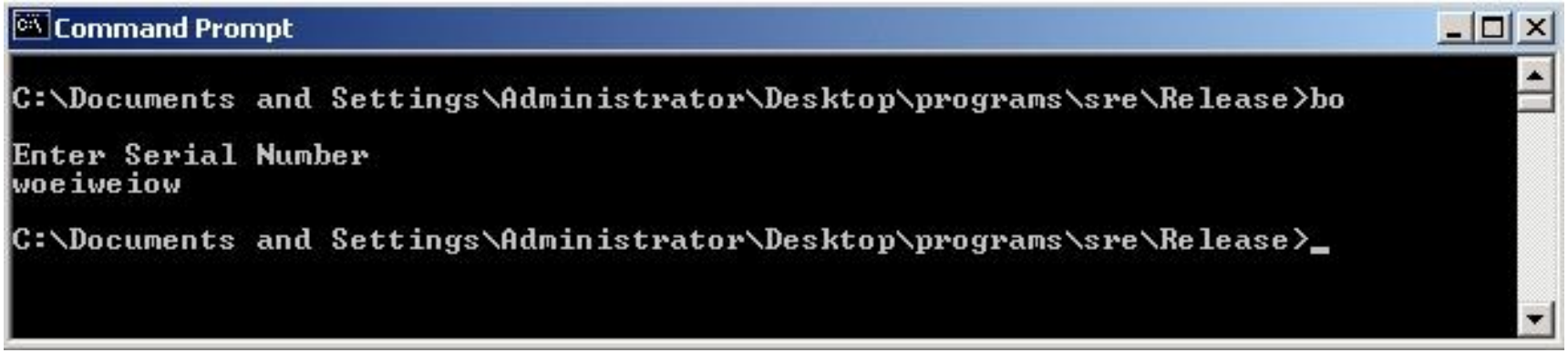
Buffer Overflow Attack due to Unsafe C/C++ Library Functions

```
void input_username(...) {
   char username[16];
   printf("Enter username:");
   gets(username);
   ...
}
```

gets does not check bounds!

Stack Overflow Attack Example

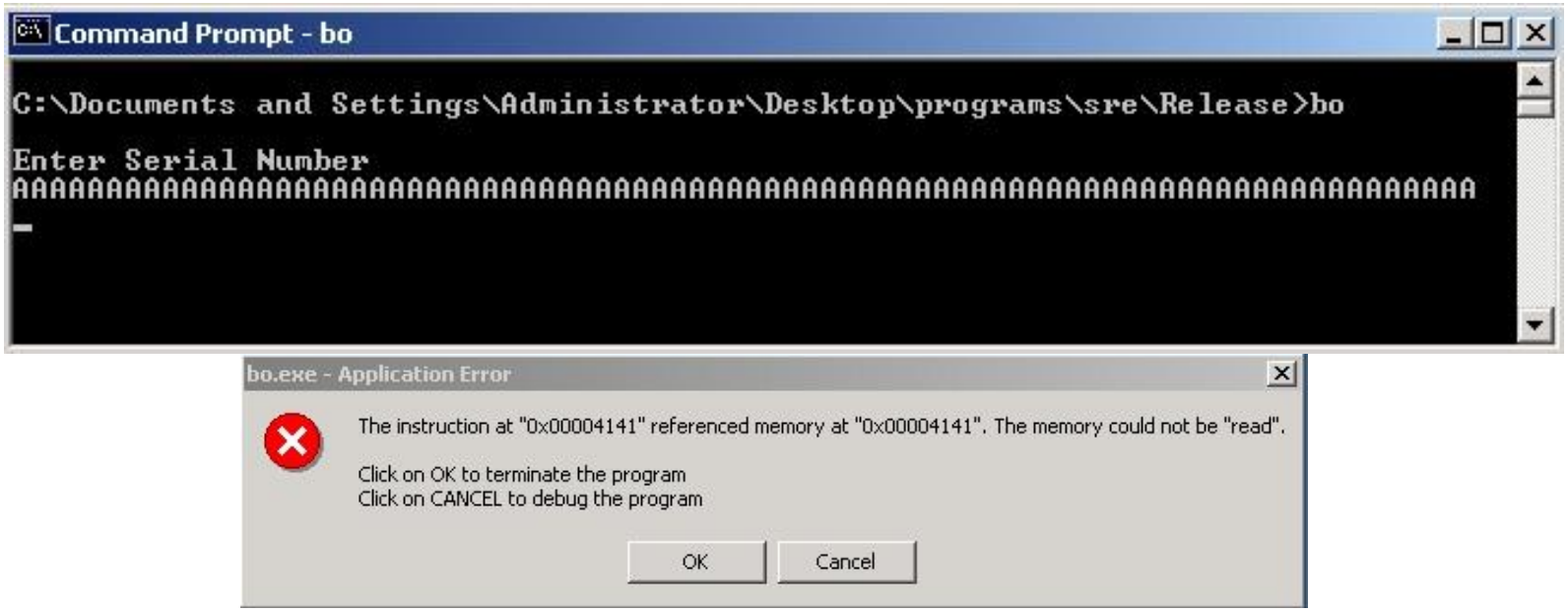
- Suppose program asks for a serial number that you do not know
- Also, you do **not** have source code
- You only has the executable (exe)



- Program quits on incorrect serial number

Buffer Overflow Present?

- By trial and error, attacker discovers apparent buffer overflow



- Note that 0x41 is ASCII for "A"
- Looks like **ret** overwritten by 2 bytes!

Disassemble Code

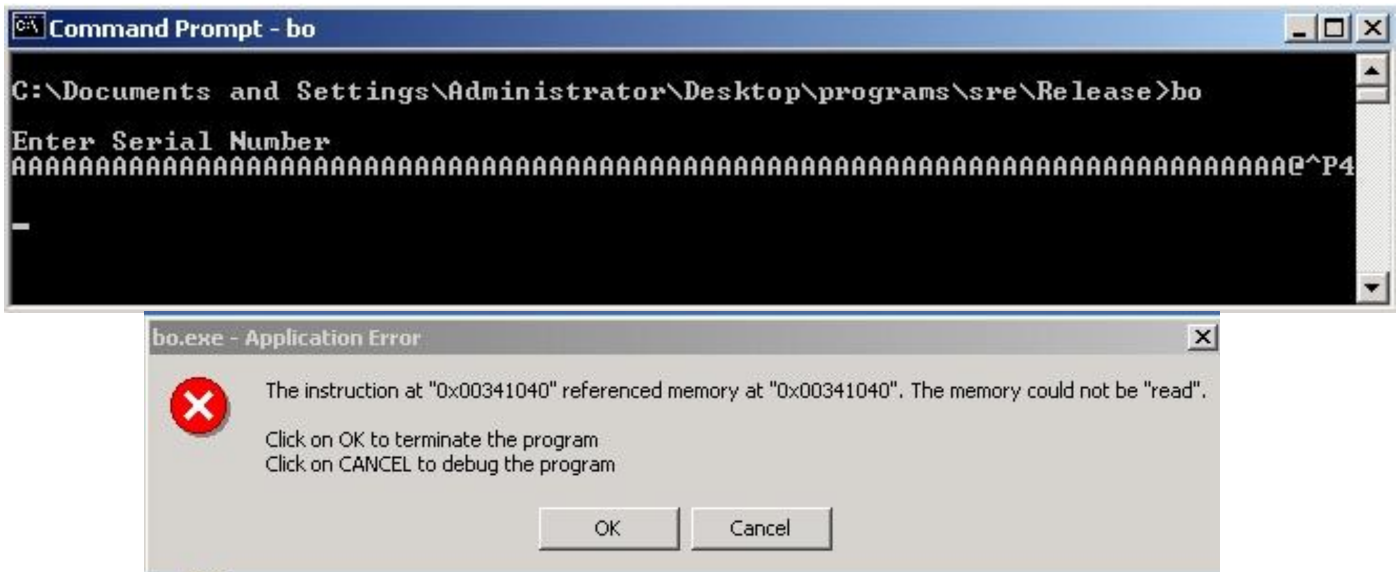
- Next, disassemble bo.exe to find

```
.text:00401000
.text:00401000                sub     esp, 1Ch
.text:00401003                push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                call    sub_40109F
.text:0040100D                lea     eax, [esp+20h+var_1C]
.text:00401011                push    eax
.text:00401012                push    offset aS       ; "%s"
.text:00401017                call    sub_401088
.text:0040101C                push    8
.text:0040101E                lea     ecx, [esp+2Ch+var_1C]
.text:00401022                push    offset aS123n456 ; "S123N456"
.text:00401027                push    ecx
.text:00401028                call    sub_401050
.text:0040102D                add     esp, 18h
.text:00401030                test    eax, eax
.text:00401032                jnz     short loc_401041
.text:00401034                push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039                call    sub_40109F
.text:0040103E                add     esp, 4
```

- The goal is to exploit buffer overflow to jump to address 0x401034

Buffer Overflow Attack

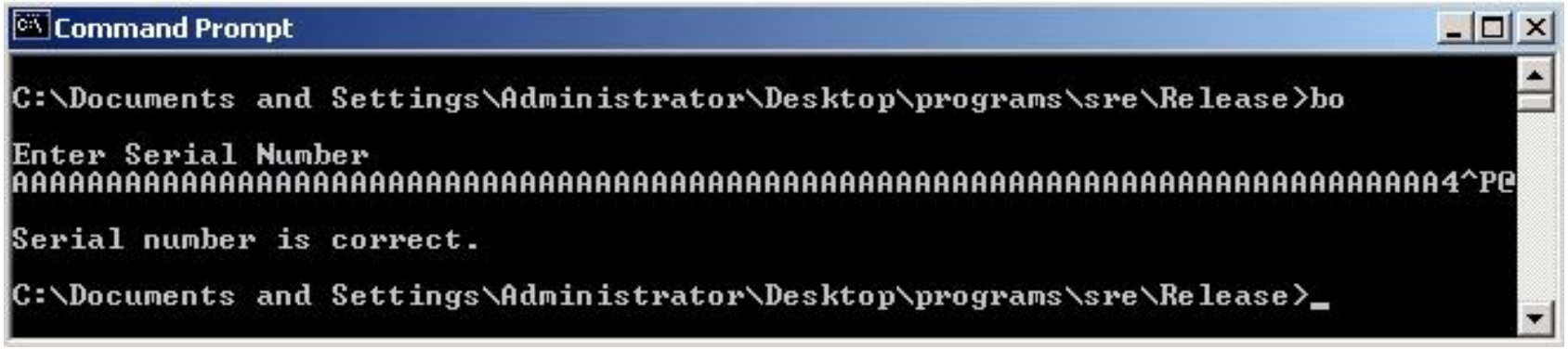
- Find that, in ASCII, 0x401034 is "@^P4"



- Byte order is reversed?
  - X86 processors are "little-endian"

Overflow Attack, Take 2

- Reverse the byte order to “4^P@” and…



- Success! We’ve bypassed serial number check by exploiting a buffer overflow
- What just happened?
  - Overwrote return address on the stack

Buffer Overflow

- Attacker did **not** require access to the source code
- Only tool used was a disassembler to determine address to jump to
- Find desired address by trial and error?
  - Necessary if attacker does not have exe
  - For example, a remote attack

Buffer Overflow Defense

- Never execute code on stack
  - W^X (write XOR Execute)
- Detect overflow
  - Use a **canary**
- Randomize stack
  - Address space layout randomization (**ASLR**)
- Don't use C/C++
  - Use **safe languages** (Java, C#)
- Use **safer C functions**
  - Additional checks on the buf length.

Marking stack as non-execute

- Basic stack exploit can be prevented by marking stack segment as non-executable.
  - W^X (write XOR Execute)
  - Support in Windows. Code patches exist for Linux.

Problems:

- Some software need writeable .text section or executable stack section
  - “self-modifying code”
  - Just-in-time compilation

Randomization: Motivations

- Buffer overflow exploits need to know the (virtual) address to which pass control
  - Address of attack code in the buffer
- Same address is used on many machines
- Idea: introduce diversity
  - Make stack addresses unpredictable and different from machine to machine

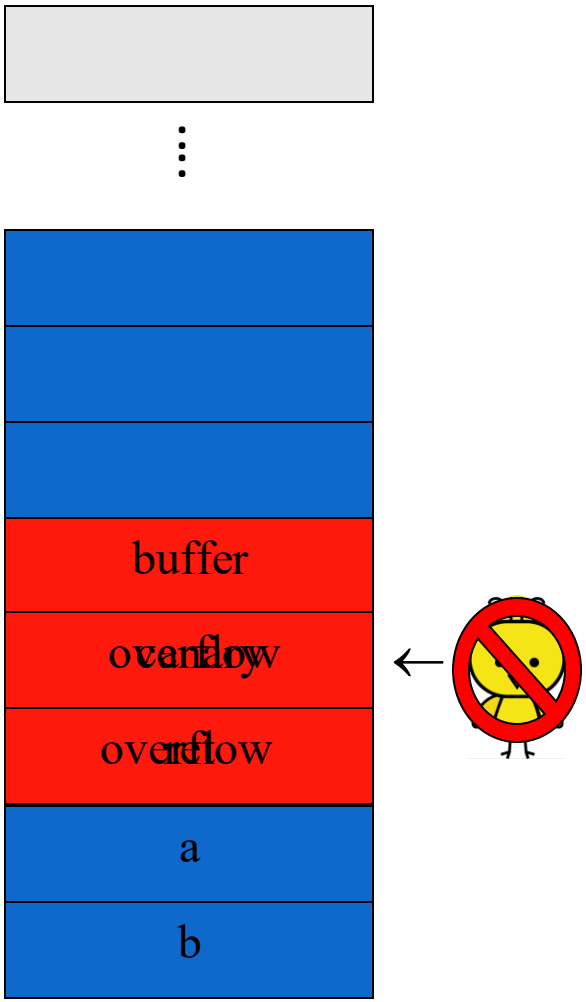
Address Space Layout Randomization (ASLR)

- Arranging the positions of key data areas randomly in a process' address space.
  - e.g., the base of the executable and position of the stack
  - Attacks:
    - Repetitively guess randomized address
    - Spraying injected attack code
- Windows has this enabled, software packages available for Linux and other UNIX variants

Buffer Overflow Defense

A simplified function stack (ignored stack frame pointer)!

- **Canary**
  - Run-time stack check
  - Push canary onto stack
  - Canary value:
    - Constant 0x000aff0d
    - Or, random value decided during runtime (see my example)



Canary

- Supported by most platforms.
  - Minimal performance effects: 8% slow down for Web servers.

- Note: Canaries don’t offer fool-proof protection.
  - Some attacks can leave canaries untouched.