Lecture 3: Buffer Overflows

presented by

Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

Introduction

- The course will work its way up from the operating system to the application layer
- We will start by looking for vulnerabilities in the way programs get executed
- The discussion will focus on general principles
  - For constructing a successful attack, details of the platform attacked need to be taken into account
  - “Old” attacks rarely work on today’s platforms

Agenda

- Background
  - Variables and buffers
  - Call stack
- Buffer overflow attacks
- Defences
  - Safe functions
  - Canaries
  - Split control and data stack
  - Data Execution Prevention
  - Address Space Layout Randomization
- Return-oriented programming

Background: C/C++

- Implementations of network protocols are often written in C/C++
  - Typical task: "serializing" of some composite data structure, i.e., package the data structure into a string, send it, unpack it at the receiver
  - Makes use of string operations
- Trade-off: performance – robustness
  - Management of memory objects is often intentionally left to the programmer for performance optimizations
  - You have to know what you are doing and be circumspect to get your code right

A very simple Web server

```c
#include <stdio.h>

int read_req(void) {
  char buf[128];
  int i;
  gets(buf);
  i = atoi(buf);
  return i;
}

int main(int ac, char **av) {
  int x = read_req();
  printf("x=%d\n", x);
}
```

```
$ ./readreq
123
x=123

$ ./readreq
1482148994124128412412 41
x=2147483647

$ ./readreq
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core
dumped)
```

What do you think has happened?

Buffer Overflows (1980s)

- Log-in in a version of Digital's VMS operating system: to log in to a particular machine, enter

```
username/DEVICE =<machine>
```

- Length of the argument "`machine`" was not checked; a device name of more than 132 bytes overwrote the privilege mask of the process started by login
- Users could thus set their own privileges

Memory Access in C/C++

- Strings are written to / read from memory
- Memory is accessed via pointers
  - Pointers are variables that hold addresses as values
- "Unsafe" write: given a pointer and a string, write to memory starting at the address pointed to until the end of the string
  - **`gets`**, **`strcpy`**, **`sprintf`**, …
  - NUL character terminates strings
  - No warning when too many characters are written!
- "Safe" write: given a pointer, a string, and a bound, write to memory starting at the address pointed to until end of string or the bound is reached
  - **`fgets`**, **`strncpy`**, **`snprintf`**, …

Variables & Buffers

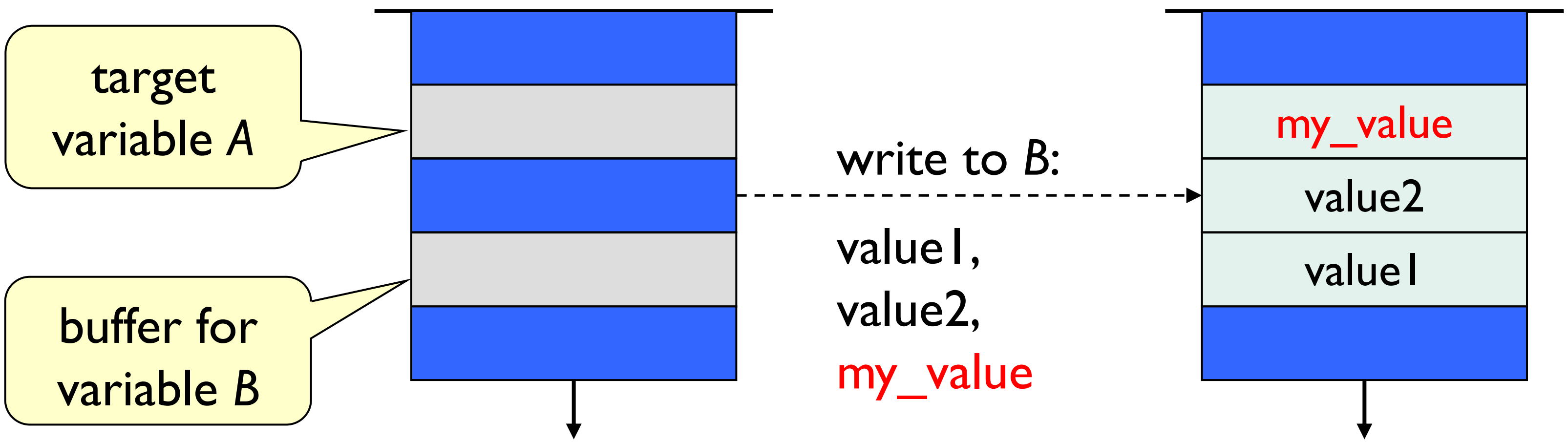
- Abstract view: programs store data in variables
- Implementation: allocate a region of memory (a.k.a. buffer) to store the value assigned to a variable
- What can go wrong when assigning a value to a variable?
  - Miscalculate position of buffer and write to a wrong location
  - Write more data than the buffer can hold

Buffer Overflows

- If the value assigned to a variable exceeds the size of the allocated buffer, memory not allocated to this variable is overwritten
- This is called a buffer overflow (overrun)
- If the memory location overwritten had been allocated to another variable, the value of that variable is changed
  - "Access via the layer below"
  - Value of a sensitive variable *A* could be changed by assigning a (deliberately) malformed value to some other variable *B*
- Depending on the location of the buffer, there are
  - Stack-based buffer overflow (covered in this lecture)
  - Heap-based buffer overflow (e.g., CVE-2021-3156: the "sudo" vulnerability)

Buffer Overflows

- Assign a value to variable *A* by writing to variable *B*
- Data written to a buffer is written upwards ↑ (towards higher addresses) from address of buffer



Buffer Overflows

- Unintentional buffer overflows crash software and have been a focus for reliability testing
- Intentional buffer overflows are a concern if an attacker can modify security relevant data
- Attractive targets are return address (specifies next piece of code to be executed) and security settings
- Type-safe languages like Java guarantee that memory management is "error-free" (more later)

The Call Stack

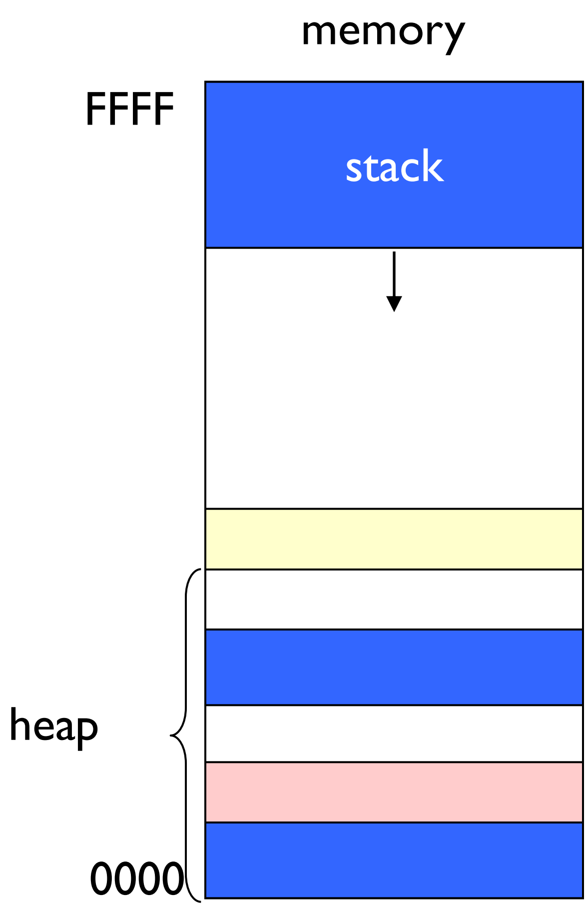
Calling and returning from functions

Function Calls

- Structured programs use functions (subroutines, methods, etc.) which may in turn call further functions
- When the runtime environment executing a program encounters a function call, it collects the data needed for executing the function in a frame, stores the frame, and starts executing the function
- Runtime environment returns to caller when the execution of the function is completed
- Function calls can be nested so frames are stored on a call stack (system stack)

Stack & Heap

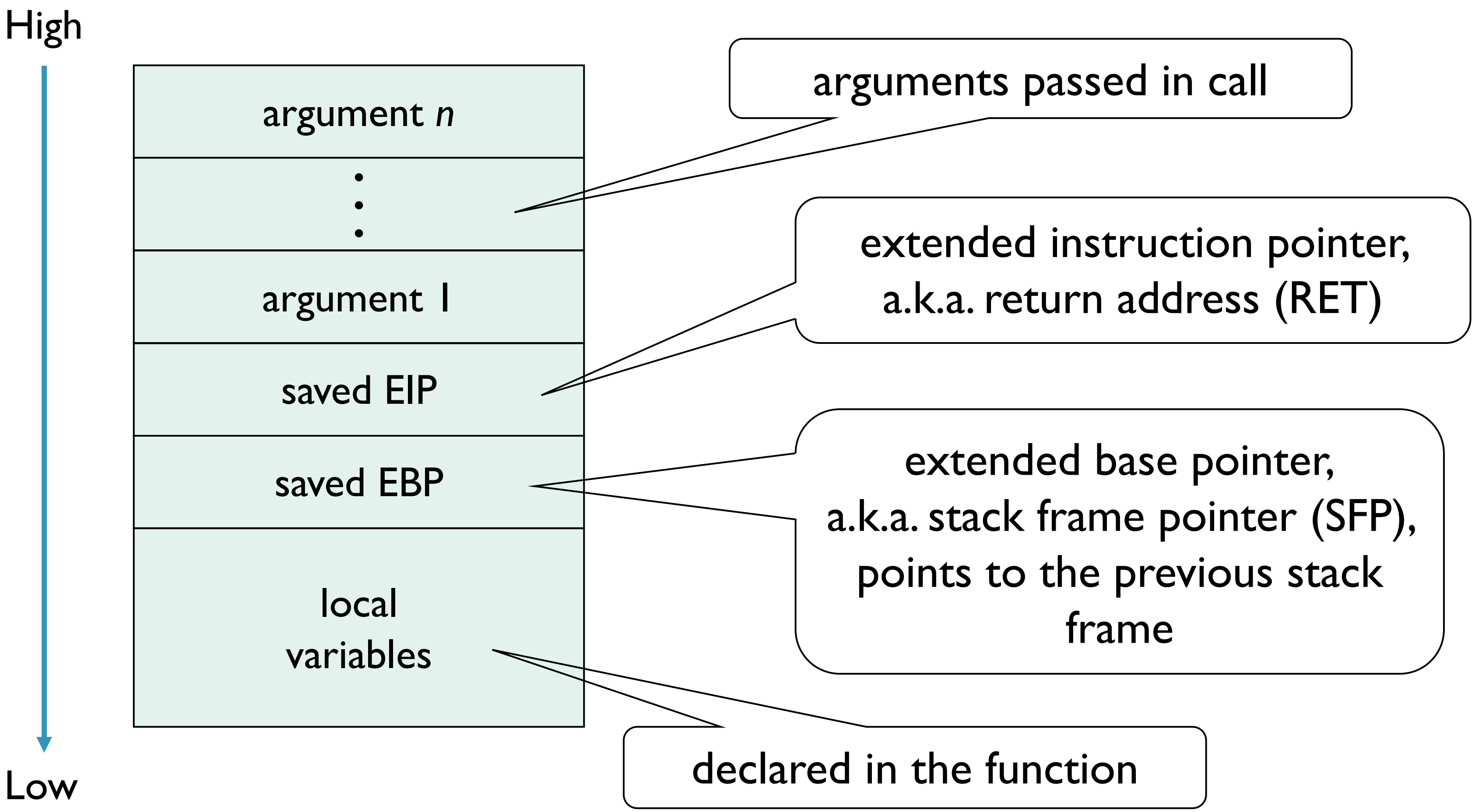
- Stack: stack frames contain return address, local variables and function arguments; relatively easy to figure out in advance where a given buffer will be placed on the stack
- Heap: dynamically allocated memory; difficulty of guessing where a given buffer will be taken from the heap depends on memory allocation scheme



Stack Frames

- Stack frame contains function arguments, return address, statically allocated buffers, and more
- When the call returns, execution continues at the return address specified
- Call stack by convention starts at the top of memory and grows downwards ↓
- Layout of stack frames is reasonably predictable (depending on compiler, operating system, …)
- Stack frame contains both user data and control data!

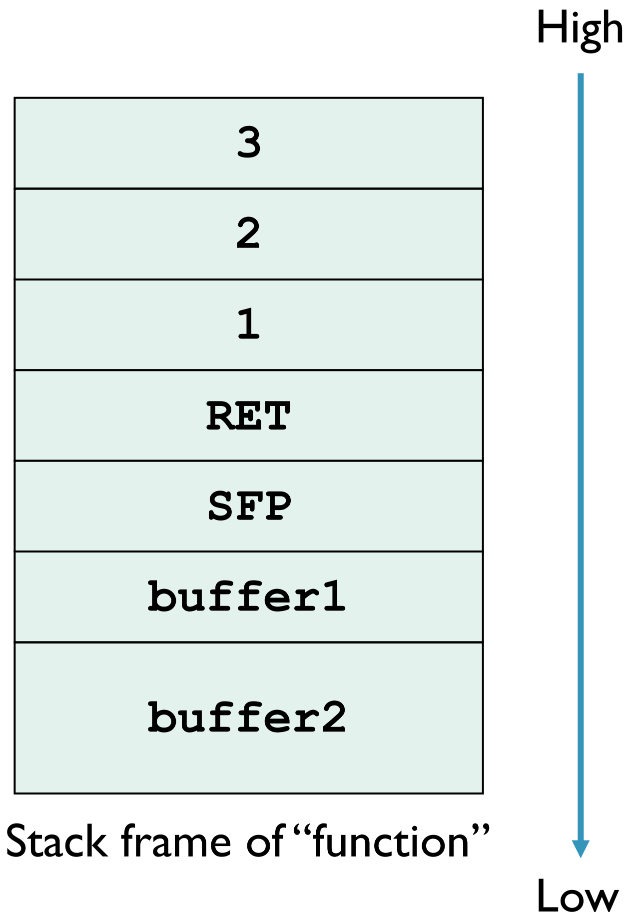
Stack Frame – Typical Layout



Stack Frame – Example

```
void function(int a, int b,
   int c)
{
    char buffer1[5];
    char buffer2[10];
}

void main()
{
  function(1,2,3);
}
```



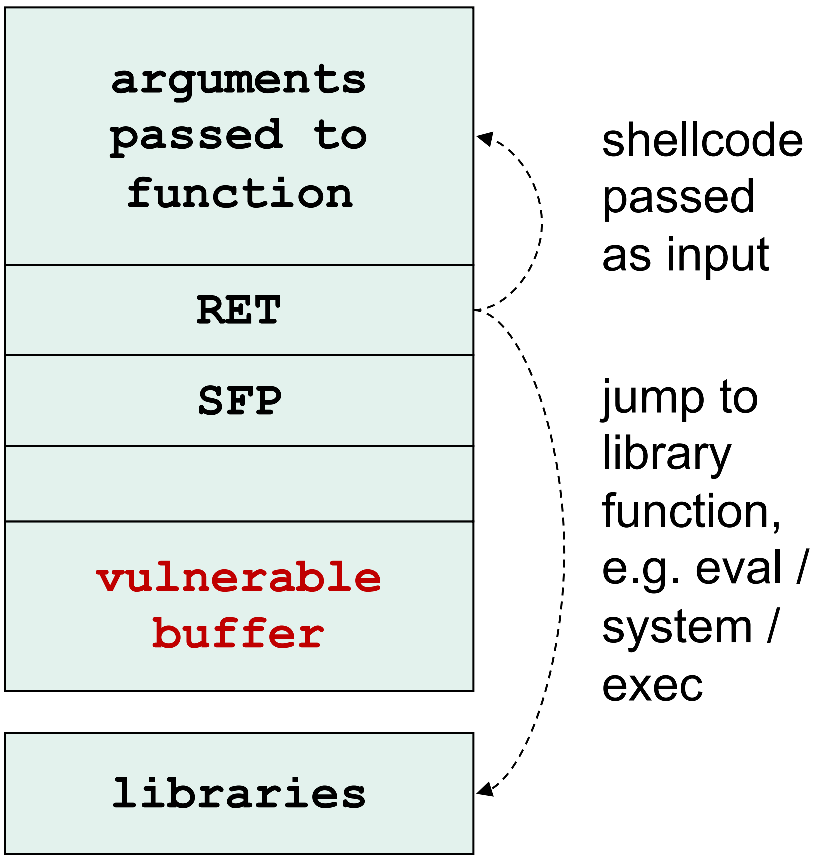
Stack frame of “function”

More examples in tutorial …

Buffer Overflow Attacks – Pattern

- Find a function that contains an "unsafe" write of user defined input to a local variable
- Attacker supplies specially crafted input to function
- Attacker's input overflows buffer allocated to that local variable until it overwrites return address to make return address point to the attacker's code
- Attacker's code commonly known as "shellcode"

Stack-Based Buffer Overflows



1. Find a buffer on the stack of a privileged program that can be overflowed
2. Overflow buffer to overwrite return address with start address of the code you want to execute
3. Jump to your code; your code is now privileged too

"Classic" Code Example

- Declare a local short string variable

```
char buffer[80]
gets(buffer);
```

  and use standard C library routine call to read single text line from standard input and save it into buffer

- Works for a line of a typical character-based terminal; corrupts the stack if input is longer than 79 characters
- Attacker loads malign code into buffer and redirects return address to the start of shellcode

Details (not discussed in detail)

More technicalities need to be considered in this and related attacks

- Attacker's malign input must not contain terminating characters; strings in C are null-terminated
- Some tricks to enter non-printable characters
- Some issues concerning little Endian and big Endian memory structures
- Attacker might not quite know where shellcode will be put; use landing pad (sledge) of NOP operations

Where to put the Shellcode?

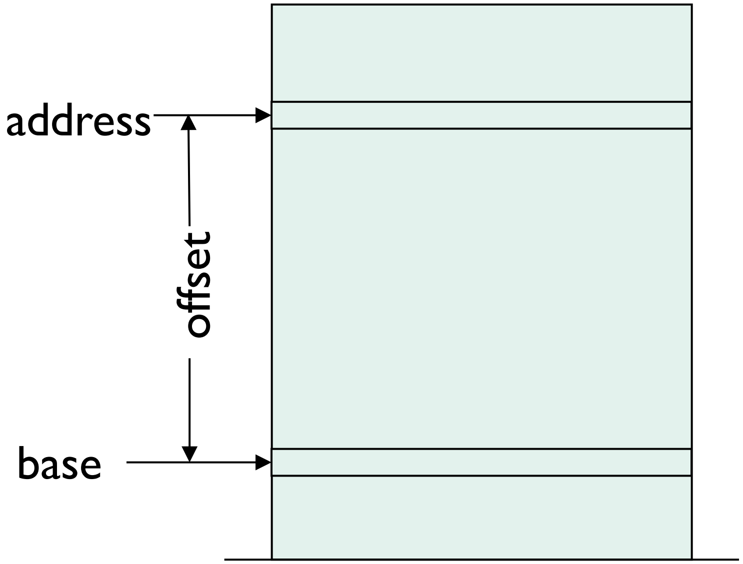
- When attack starts, the system isn't corrupted yet
- argv[]-method: put shellcode on the stack as part of the malign input
  - Attacker has to guess distance between return address and address of the argument containing the shellcode
  - Detailed examples in "Smashing the Stack for Fun and Profit"
- return-to-libc-method: jump to `eval` library function that will execute commands provided as user input
- Return-oriented programming (ROP): jump to code segment in some existing executable

Detour – Relative Addressing

- Note: attacker does not have to guess the absolute address of the shellcode, only its relative address
- Relative addressing: compute

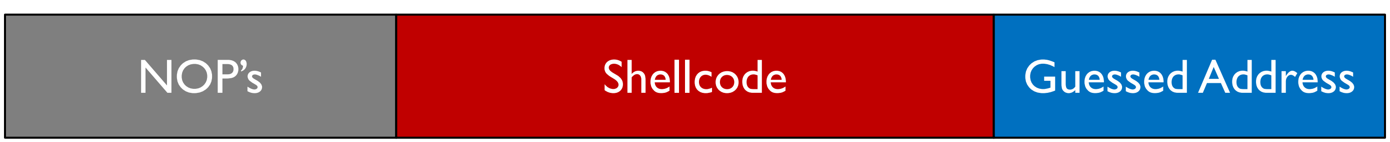
  $actual\ address = base + offset \times size$

- When addressing an element in an array, $base$ is the start address of the array, $offset$ is the array index (multiplied by the size of array elements)
- Used by compilers as it is usually not known at compile time where code will be loaded in memory

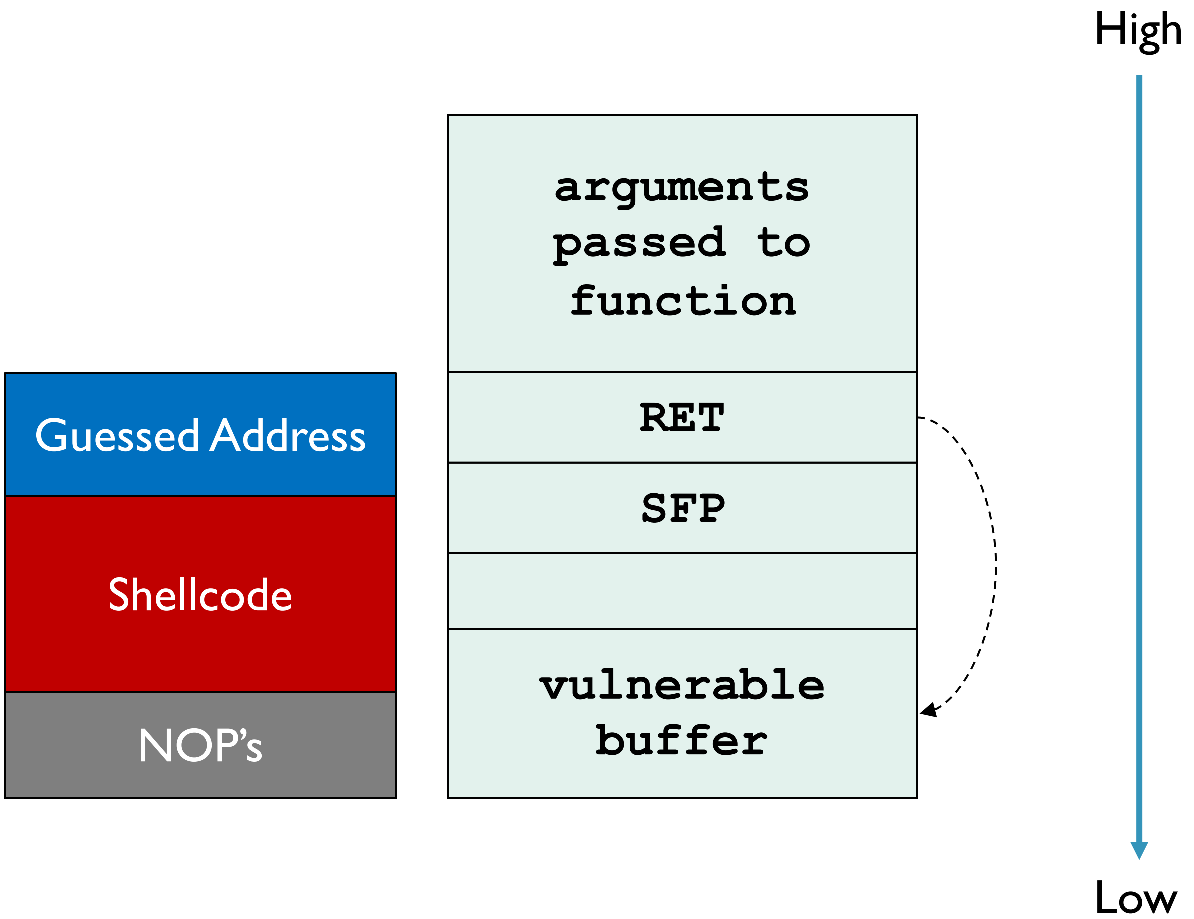


Putting it all together

- Now assume we (the attacker) have a program that does a string copy from an input buffer that we control, how do we use this shellcode?
  - We can control where the program will return to by overwriting the return address, but we don't know where the shellcode will sit in memory, so we have to guess
  - We put our guesses at the end since it's the end of the buffer that will overwrite the return address
  - We pad the front of the shell code with NOP's (landing sled). This way if we jump into any address in that region, we will eventually execute the shellcode



Putting it all together



Internet Worm of 1988 (Morris)

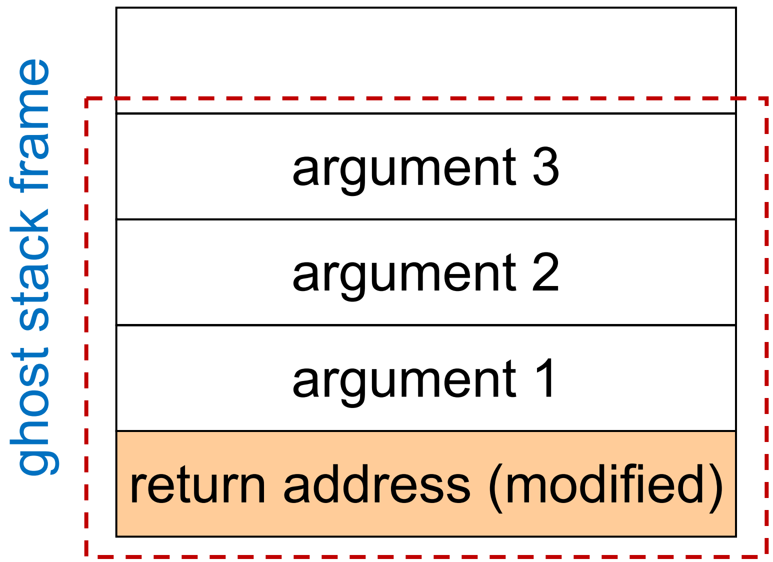
- Sent special 536 byte message to overflow a buffer in the fingerd daemon and overwrite the system stack:

```
pushl   $68732f      push '/sh <NUL>'
pushl   $6e69622f    push '/bin'
movl    sp, r10      save address of start of string
pushl   $0           push 0 (arg 3 to execve)
pushl   $0           push 0 (arg 2 to execve)
pushl   r10          push string addr (arg 1 to execve)
pushl   $3           push argument count
movl    sp, ap       set argument pointer
chmk    $3b          do "execve" kernel call
```

- On return to `main`
  - `execve("/bin/sh",0,0)` is executed, opening a connection to a remote shell via TCP

- Video: https://youtu.be/xdnwR_T-qx0?t=262

return-to-libc



- Return address changed so that execution returns to a library function
  - Library function expects its arguments on the stack
- Attacker puts arguments to library function in a ghost stack frame
- Defence: library functions that take arguments only from CPU registers

Defending Against Buffer Overflow

Closing the loopholes

Countermeasures

- Can be classified in terms of their “locations”
  - Programming language
  - Libraries
  - Compiler
  - Hardware
- Can be classified in terms of security strategies
  - Prevention: stack overflows cannot occur
  - Detection: stop execution when a stack overflow is detected
  - Mitigation: consequences of a stack overflow are made less serious

Countermeasures

- We will now systematically analyze how a buffer overflow attack can be stopped
- Each defence can help if the previous ones had not been effective:
  1. Check input arguments for variables / arrays so that input sizes fit the memory allocated
  2. Protect return address
  3. Do not allow executable inputs
  4. Make it difficult to guess location where shellcode / system libraries will be placed

Prevention – Programmer

- When programming in C or C++, the first line of defence against buffer overflow is the programmer
- C is infamous for its unsafe string handling functions: **`strcpy`**, **`sprintf`**, **`gets`**, …
- Example: **`strcpy`**

```
char *strcpy( char *strDest,
              const char *strSource );
```

  - Throws exception if source or destination buffer are null
  - Undefined if strings are not null-terminated
  - No check whether destination buffer is large enough
  - http://www.cplusplus.com/reference/cstring/strcpy/

‘Safe’ Functions

- Replace unsafe string functions by functions where the number of bytes/characters to be handled are specified: **strncpy**, **_snprintf**, **fgets**, …
- Example: **strncpy**

```
char *strncpy(char *strDest, const char
*strSource, size_t count );
```

- You still have to get the byte count right

- http://www.cplusplus.com/reference/cstring/strncpy/

Using ‘Safe’ Functions

- You have to get your arithmetic right
  - Problem will be discussed in next lecture
- You have to know the correct maximal size of your data structures
  - Straightforward for data which are used within a single function
  - May be tricky for data structures shared between programs
  - If you underestimate the required length of the buffer your code may crash

Guards – Example

```
bool HandleInput_Strncpy1( const char* input)
{
   char buf [80];
   if (input == NULL) {
        assert(false);
        return false; }

   strncpy (buf, input, sizeof(buf) - 1);
   buf[sizeof(buf) - 1] = '\0';
   // more processing …
   return true;
}
```

Problem: if input is too long it will be truncated; this might cause problems elsewhere

Guards – Example

```
bool HandleInput_Strncpy2( const char* input)
{
   char buf [80]
   if (input == NULL) {
        assert(false);
        return false; }

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

   strncpy (buf, input, sizeof(buf));

   if(buf[sizeof(buf) - 1] != '\0';
        { return false;}  //Overflow!

   // more processing …
   return true;
}
```

strsafe.h

- **strsafe** header for un-defining unsafe functions, e.g.

```
#define strcpy unsafe_strcpy
```

  - Errors raised at compile time when code contains this unsafe function
- String handling library that is true to the abstraction
  - No overflow of destination buffer
  - Buffers guaranteed to be null-terminated
- Question: why can't we get rid of all buffer overflows by replacing **strcpy**, **sprintf**, **gets** and the like?

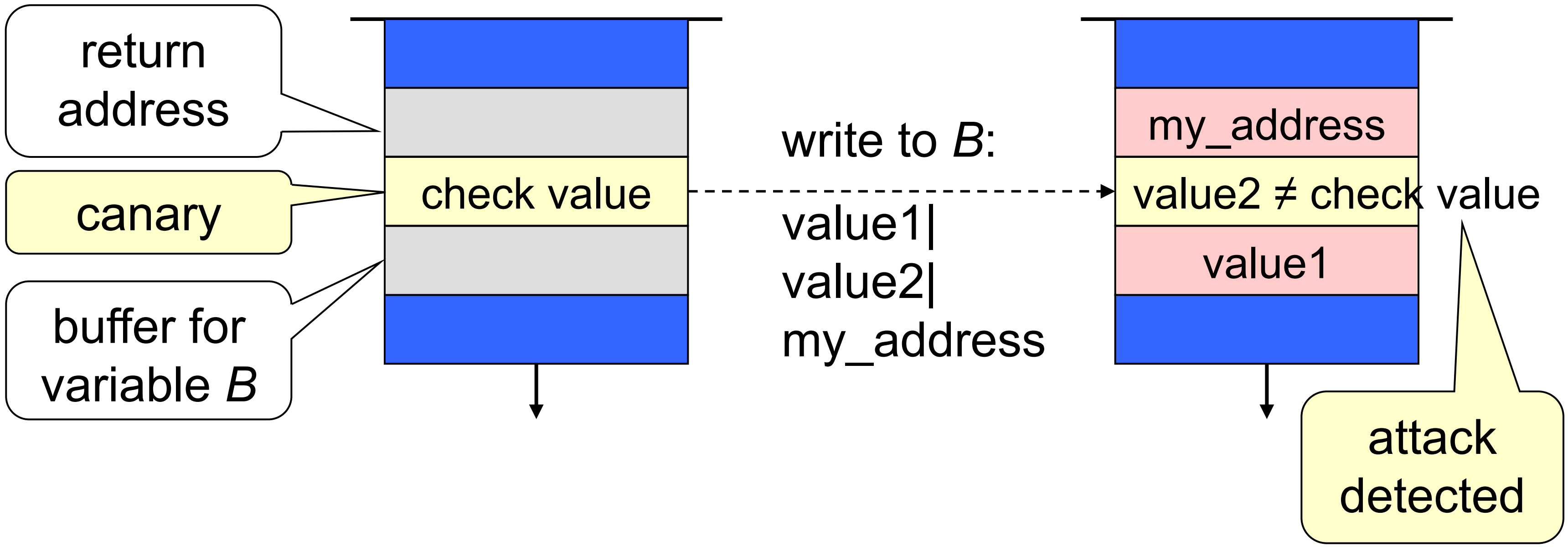
Type Safe Languages

- Defence: use a programming language where buffer overflows cannot happen "by design"
- Safety guaranteed by static checks and by runtime checks
  - Automatic array bounds checking
  - Automatic garbage collection
  - Programmers can't make mistakes when managing memory
  - Programmers cannot optimize memory usage
  - Programmers cannot themselves take care of multiple copies of sensitive data (e.g. a password)
- Examples: Java, Ada, C#, Perl, Python, etc.
- More on type safety later in the course

Detection – Compiler

- Detect attempts at overwriting return address
- Compiler places a check value ('canary') in the memory location just below the return address
  - The term canary is borrowed from coal mining
- Before returning, check that canary hasn't changed
- Stackguard: random canaries
  - Alternatives: null canary, terminator canary
- Source code has to be recompiled to insert placing and checking of the canary

Canaries



return address
canary
buffer for variable B
check value
write to B:
value1|
value2|
my_address
my_address
value2 ≠ check value
value1
attack detected

Remark on Check Values

- (Integrity) check values are a generic defence
- Security requirements:
  - Check values must not be predictable
  - Reference values must be integrity protected
- Two options for integrity protection:
  - Write to protected memory
  - Calculate checksums using a secret cryptographic key (magic value); only the key has to be protected; no need to write the key when compiling a program

Non-Executable Memory

- Support at hardware level for marking memory as non-executable
  - AMD: NX (no execute) bit since Athlon 64
  - ARM: XN (eXecute never)
  - Intel: XD (execute disable bit, EDB)

Data Execution Prevention

- Data Execution Prevention (DEP): mark memory location a process has written to as non-executable
- W⊕X (W^X if you are a C programmer) protection
  - Memory can be writeable or executable, but not both
  - Shellcode placed by attacker will not be executed
- This approach cannot be applied to components creating executable code, e.g. compilers
  - Exploited by JIT spraying attacks

Randomizing Memory Allocation

- Stack buffer overflow attacks have to know the position of a target buffer in relation to return address, and the approximate location of attack code
- Random changes to memory allocation can reduce impact of buffer overflow attacks
- Example: Address Space Layout Randomization (ASLR) in Linux, Windows, …
  - Randomizes memory location, e.g. of stack, heap, libraries
  - Defence against return-to-libc attacks

Lesson

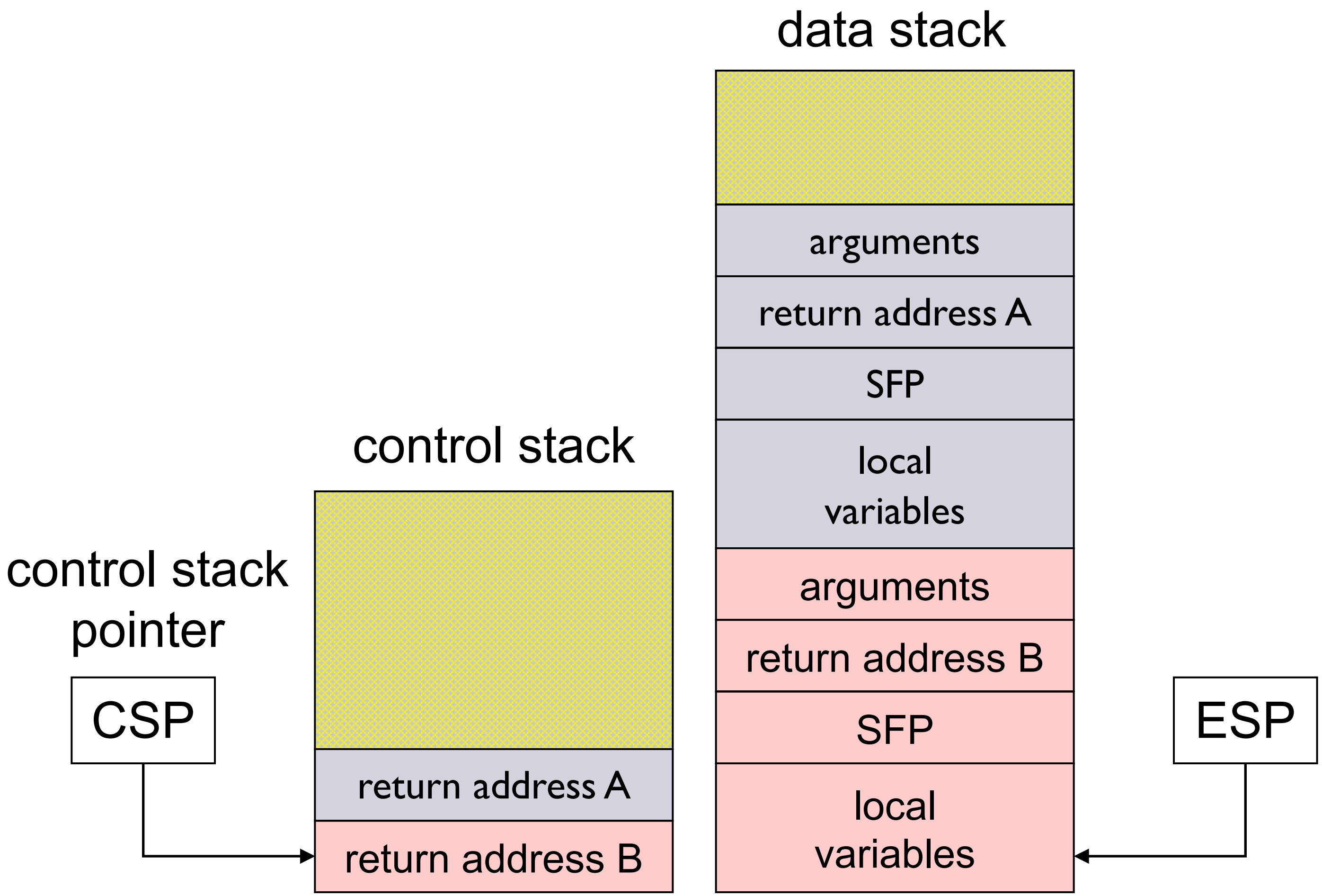
Randomness is important for security

Split User Data from Control Data

- Stack-based buffer overflow attacks overwrite return address
- Such attacks would be blocked if the return address is not taken from the stack but from a location the user (input) cannot touch
- Abstraction: separation of control and user data
- Control stack: memory area separate from data stack
- Implementations in hardware and software have been proposed in the research literature

From: Xu et al. Architecture Support for Defending Against Buffer Overflow Attacks

Split Control and Data Stack



Return-Oriented Programming

A smart way around Data Execution Prevention

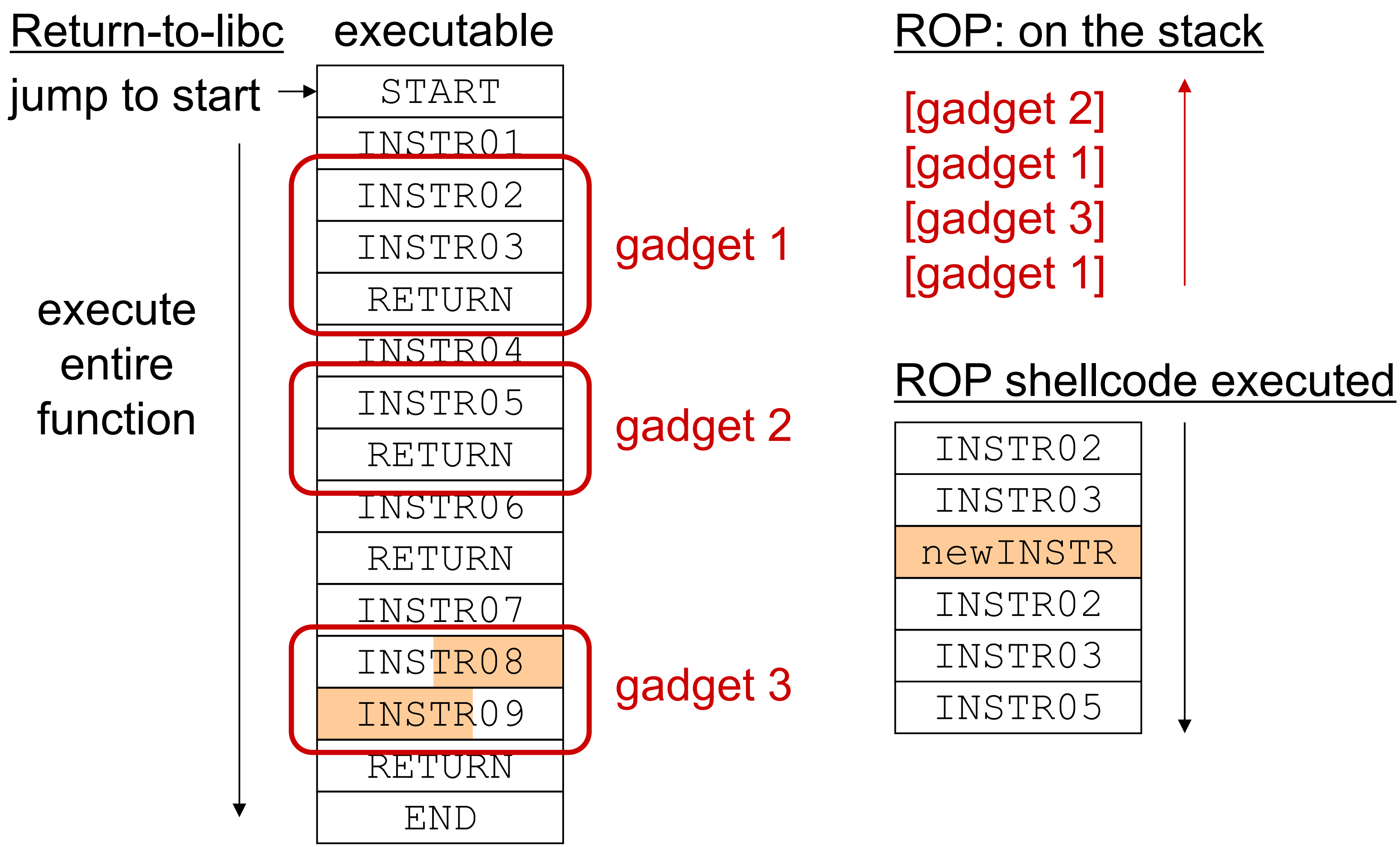
Bypassing DEP

- Shellcode need not be inserted; the attacker may use existing executables, e.g. from system libraries, in return-to-libc attacks
- Attacker is limited to given executables
- Can one use executables in unintended ways, e.g. by selectively using instruction blocks from executables?

Return-Oriented Programming

- Attacker uses short code segments in executables, so-called gadgets, that end with a return command
  - When misaligned memory access is permitted the attacker may find byte sequences starting in the middle of a word that constitute valid machine instructions
  - Gadgets serve as building blocks for writing shellcode
  - If enough gadgets are found in the code base, shellcode with arbitrary functionality can be built ($\rightarrow$ weird machine)
- Stack buffer overflow attack puts gadget addresses on stack; after returning from one gadget, the attack jumps to the next (call stack used as a trampoline)

Return-to-libc and ROP



Return-Oriented Programming

- ROP more powerful than return-to-libc attacks
- Initially assumed to be only an issue in architectures with variable length instructions
- Later shown that such vulnerabilities can also exist in RISC architectures (fixed length instructions)
- Real exploits have been documented
  - AVC Advantage voting machine
  - Adobe Reader 9.3 with DEP enabled
- Similar attack patterns exist where gadgets need not end with a return (jump-oriented programming)
- Tutorial Video: https://www.youtube.com/watch?v=zaQVNM3or7k

Resources

- Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls, Proceedings of CCS 2007, pages 552–561
  - http://cseweb.ucsd.edu/~hovav/papers/s07.html
- Erik Buchanan, Ryan Roemer, Hovav Shacham, Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC, Proceedings of CCS 2008, pages 27–38
- Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, Hovav Shacham. Can DREs Provide Long-Lasting Security? USENIX 2009
- Jduck. The Latest Adobe Exploit and Session Upgrading, 2010
  - https://blog.rapid7.com/2010/03/18/the-latest-adobe-exploit-and-session-upgrading/

Comments and Further Countermeasures

Growing the Stack

- Comment in Paul A. Karger & Roger R. Schell: Thirty Years Later: Lessons from the Multics Security Evaluation
- Stack buffer overflows occur when stack grows downwards but data are entered upwards into a buffer
- Such problems would not occur if the stack starts at the bottom of memory and grows upwards

Targets

- Overwriting the return address is a common form of a buffer overflow attack
- If return address cannot be reached, alternative targets include:
  - overwrite a function pointer variable on the stack
  - overwrite security-critical variable value on stack
  - overwrite previous frame pointer
  - overwrite arguments with ghost stack frames

No Silver Bullet

- None of the countermeasures are perfect
- The earlier buffer overflows are addressed in the design process the better
- Systematic work on removing security relevant buffer overflows started 15-20 years ago
- Prediction (Jon Pincus, 2004): buffer overflows will disappear as an issue in the next 1-2 years, but there will be other software security issues
  - This prediction has by and large held up; major problems today are cross-site scripting, SQL injection, …

Conclusions

- Aleph One's paper on buffer overflow attacks focused general attention on software security
- Main design problems exploited:
  - Unsafe write operations
  - Stack frame contains both user and control data
- Significant progress has been made since, both in research and in the defences deployed in the wild
  - Stack buffer overflow attacks hardly work in today's operating systems
- Attackers had to become much more sophisticated
  - More on this in the coming lectures

Tutorial

ASLR & Jump-Oriented Programming

- Collect information on ASLR in Linux and in Windows
  - What is being randomized? How random is randomization?
- Jump-oriented programming creates its own trampoline for linking gadgets instead of using the call stack
  - Jump-Oriented Programming: A New Class of Code-Reuse Attack
- Read up on ASLR and JOP; you should be able to explain the fundamentals of these attack methods