

ENTS 689i
Part II: System Security

Lecture 5: Application Security

Part II: Goals

- Learn how systems work and break
- Focus on individual system protection
- Understand attacker methods and mechanisms
- Study modern defense mechanisms
- **Develop a security mindset**

Part II: Organization

- Format is same as Part I:
 - Three lectures
 - In-class exam, followed by mini lecture
- In addition, there will be two hands-on labs
- Student interaction is encouraged
 - Try examples at home
 - The more you do, the more you will learn
- Office hours immediately following class

Ethical Computing

- We are going to discuss concepts and techniques that work against real systems
- Isolated virtual machines will be used for all experimentation
- **YOU** are responsible for knowing what is acceptable and what is not

Part II: Outline

- October 9: Application/program security
 - October 10: HW 2 out
- October 16: Operating system security
 - October 17: HW 2 due, HW 3 out
- October 23: Malicious software (malware)
 - October 24: HW 3 due
- October 30: Forensics and response
 - EXAM 2

Part II: Outline

- October 9 (Today)
 - System security intro
 - Application security basics
 - Low-level programming bugs and exploitation
 - Mitigating low-level programming bugs

Motivation

- Why study system security?
- We rely on computers for all aspects of life
- When they fail, things can go bad
- Years ago, reliability/security didn't matter
- Now, system failure can result in loss of money and/or life

Context

- Security is never the primary goal
 - We don't live life to be secure
 - We require security in order to pursue our goals
- The same is true for systems
 - If you won't/can't be attacked, no need to harden your system
 - Spend your resources elsewhere
- But real systems do get targeted and do fail

What is security?

- In the eye of the beholder
- Generally, “bad things not happening”
- Different from reliability?
- What kinds of bad things can happen?
 - Loss of secret/private information
 - Your machine used to attack others
 - Your machine no longer works
- NOT a binary property
- One view: security is **risk management**

Why do systems fail?

- Not designed with security in mind
- Invalid **assumptions** or **threat model**
 - System will be used in physically secure environment
 - System will be single-user
 - User won't give away his password
 - User is trustworthy
 - Implementation will be correct

Why do systems fail?

- Complexity abounds
- Many people involved
 - Designers
 - Developers
 - Administrators
 - Users
- Many parts of the system
 - Hardware
 - Software

Why do systems fail?

- Security is often bypassed
- Must be integrated with the whole



Why do systems fail?

- Because they have bugs!
- Many kinds (and many ways to classify)
 - Improper input validation
 - Bad bounds checking
 - Insufficient/broken authentication
 - Memory management errors (e.g., double free)
 - Race conditions
 - Data type conversion errors
 - Poor exception handling

Software lifecycle

- Each phase plays a role in security
- Bugs can happen at any step
- Solutions/strategies exist for each
- What are some examples?

Requirements

Design

Implementation

Testing

Deployment

Common terminology

- **Flaw**: defect in a program (bug)
- **Vulnerability**: flaw that leaves a system open to being compromised, or **exploited**
- **Error**: human mistake that caused the flaw
- **Fault**: incorrect “step” by the program/machine
- **Failure**: system does not perform as required

Software problems

- Happen all of the time
- Accidental
 - Mars Lander conversion error
 - Year 2000 issue
- Adversarial
 - Volumes of malicious software on the Internet
 - Millions of dollars in defensive software tools
 - Large networks of zombie hosts (botnets)
- Current overall state: not very good

Low-level software bugs

Software bugs

- Programmers make mistakes
- Consequence of bug depends on nature of mistake and details of underlying system
- Adversary can combine knowledge of bug and system to achieve malicious goal

Software exploitation

- Attacker's goal: make software do something not intended by the programmer
- Requires manipulating inputs
 - Bad format of single input
 - Multiple good/bad inputs in sequence
- Puts software in unexpected state
- Violates program's **integrity**

Low-level programming bugs

- Some bugs happen because programmer's abstraction is not enforced
- Array size
 - Buffer overflows
 - Off-by-one errors
- Integer size
 - Integer overflows
- Missing/extra parameters
 - Format string bugs
- Referencing uninitialized data

Unsafe languages

- Low-level languages like C allow direct manipulation of underlying data
- Particularly, the compiler allows things like:
 - Arbitrary type casts (including functions)
 - Arbitrary memory/pointer accesses
 - Unbounded array accesses
- Flexible and powerful
- But with power comes danger

Unsafe, but valid C

```
int main()  
{  
    struct circle mycirc;  
    struct square *mysquare;  
    mysquare = (struct square *)&mycirc;  
    mysquare->sidelen = 4;  
}
```

- Arbitrary cast/manipulation

More unsafe C

```
int main()  
{  
    int myarray[10];  
    myarray[30] = 10;  
}
```

- Access beyond array bounds

C library

- Implementation of standard API
- Contains a number of unsafe functions
- Now deprecated, but exist on most systems
- **gets ()**
 - read input until EOL or EOF
- **strcpy ()**
 - copy string until NULL terminator
- **strcat ()**
 - concatenate until NULL terminator

Buffer overflow history

- **Morris worm (1988)**
 - Buffer overflow in fingerd network service
 - Also used sendmail misconfiguration
 - Due to bug in worm(!), overwhelmed machines
- **Code Red (2001)**
 - Buffer overflow in IIS
 - Over 350,000 machines in 14 hours
- **SQL Slammer (2003)**
 - Overflow in MS SQL server
 - 75,000 machines in 10 minutes

What is a buffer overflow?

- As name suggests, filling beyond end of a “buffer” -- a contiguous piece of memory
- Common bug in I/O routines
- User supplies data, programmer assumes length or improperly verifies it
- What's so interesting about that?
 - *Something* must come after that data. What?
 - Depends on the **architecture** and **compiler**

Simple example

```
int main()
{
    char buf[20];
    printf("Fill my buffer!\n");
    gets(buf);
    printf("You said %s\n",buf);
}
$ gcc -o example example.c
```

Simple example

```
$ ./example
```

```
Fill my buffer!
```

```
hi my name is nick
```

```
You said hi my name is nick
```

```
$ ./example
```

```
Fill my buffer!
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
You said AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Segmentation fault
```

C/x86 compiler conventions

- Locals (e.g., buf[]) are stored on stack
- Stack also used for call/return
 - C compiler pushes parameters on stack
 - call instruction pushes return address on stack
 - return instruction pops return address, jumps
 - calling function find return value on stack
- x86 stack grows **down**
- esp register points to “top” of stack

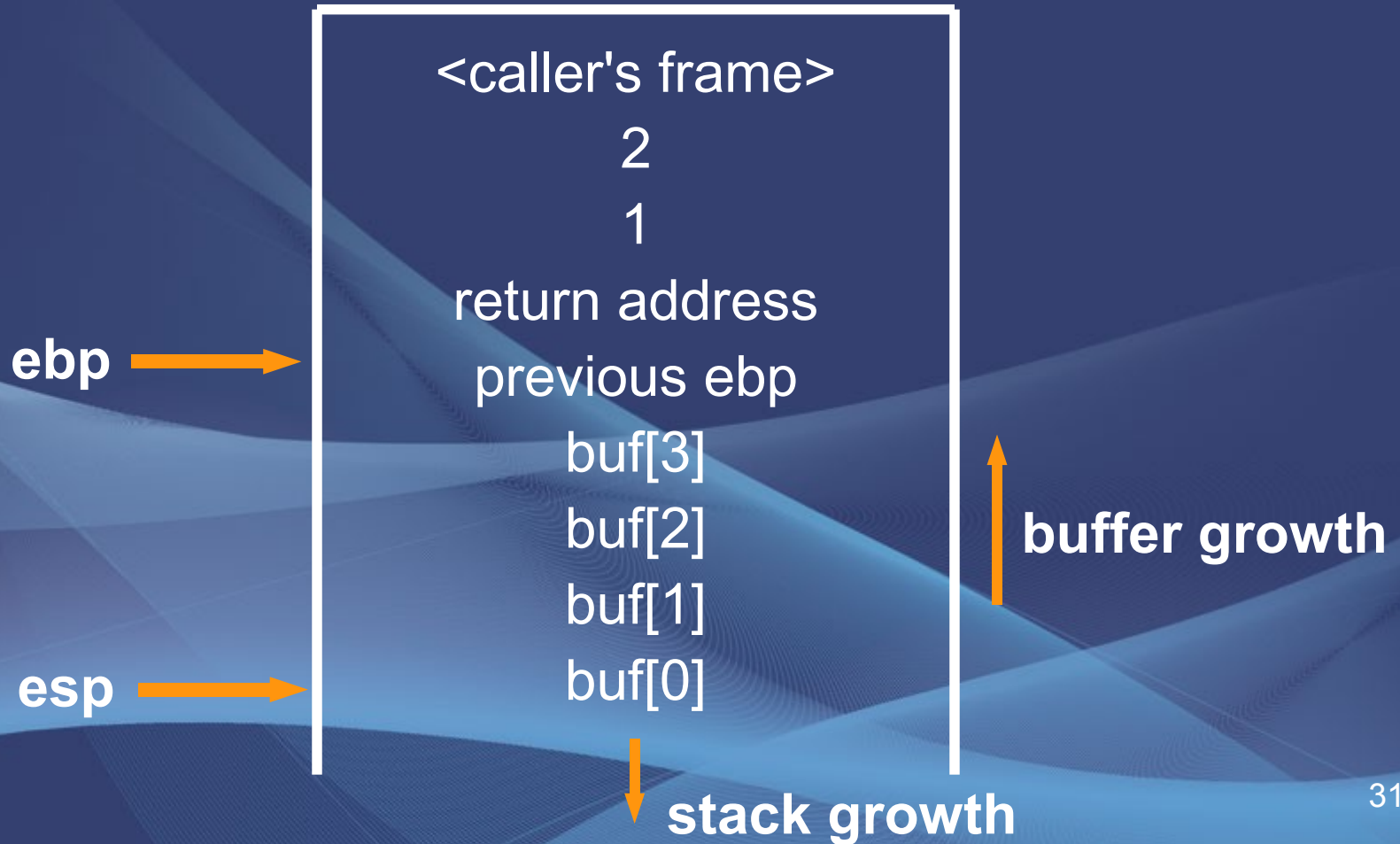
funcall example

```
int foo(int a, int b)
{
    char buf[4];
    gets(buf); ← unsafe
}

int main(int argc, char *argv[])
{
    foo(1,2);
}
```

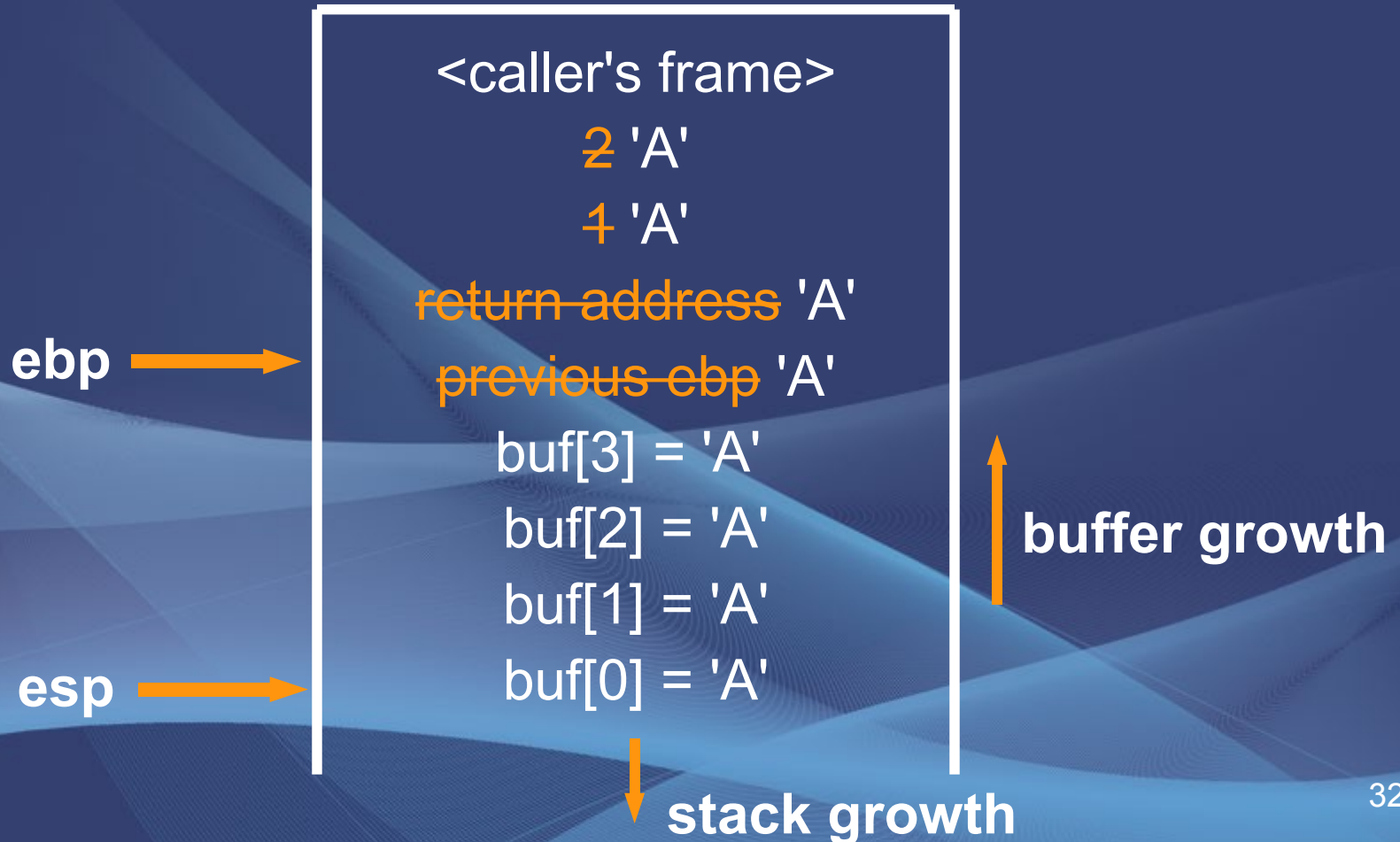
funcall example

- Stack entering foo(1,2)'s frame



funcall example

- Stack just after return from gets()



funcall example

- Look what we overwrote!
- What if we replaced 'A' with something else?
- Easy way to make control flow changes!
- Referred to as “gaining execution”



<caller's frame>

2 'A'

4 'A'

~~return address~~ 'A'

~~previous ebp~~ 'A'

buf[3] = 'A'

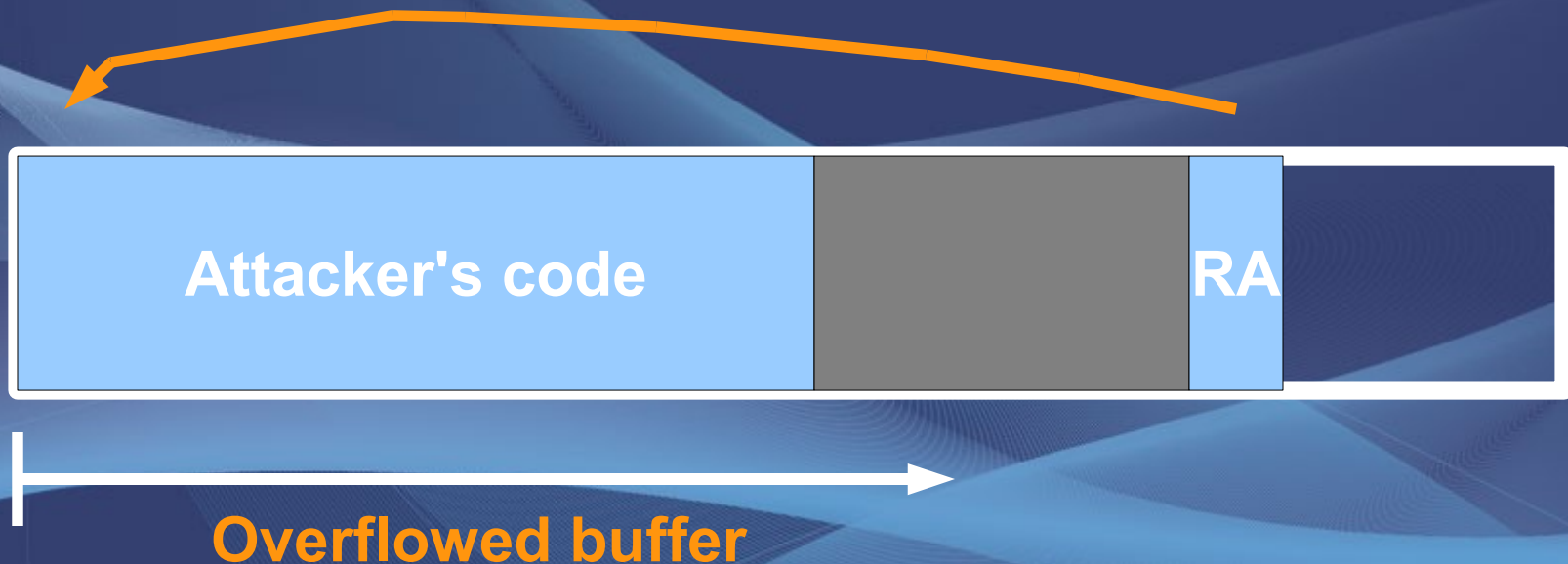
buf[2] = 'A'

buf[1] = 'A'

buf[0] = 'A'

Stack Overflows

- Where to redirect execution?
- Attacker can inject new code, jump to that
 - Inserted into overflowed buffer (or other buffer)
 - Commonly called “shellcode”



Attacker challenges

- Insufficient buffer space
 - Usually can get by with very small amount
- Need to know address of target buffer
 - Can guess and pad with nop instructions (commonly called nop sled)
- Overflowed data may be filtered
 - No NULL characters, other types of terminators
 - x86 ISA is complex: many choices for functional equivalence

Return-to-libc

- Attacker does not have to inject code to modify execution
- Can jump to existing program code instead
 - e.g., `exec()` system call with supplied arg
- Can jump to existing data as well
- Anything that decodes to valid target instructions!
- See Shacham's work on Return Oriented Programming (BH 08, ACM CCS 07/08)

Variations of Stack Smashing

- Look for other valuable data on stack
- Function pointers
- Base pointer (ebp), instead of return addr
 - How does this help?
- C++ exception pointers
- Data values that cause desirable paths
 - E.g., skip authentication/authorization checks

Heap overflows

- Buffers don't have to be on stacks
- Dynamically allocated data go into *heap*
- Typically memory obtained via `malloc()`
- No return address/frame pointer
 - Not directly used in control-flow like stack
 - Can target function pointers, other data

Heap overflow example

```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar( vulnerable* s, char* one,
char* two )
{
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

Heap overflow example

```
typedef struct _vulnerable_struct
```

```
{
```

```
    char buff[MAX_LEN];
```

```
    int (*cmp)(char*,char*);
```

```
} vulnerable;
```

Overflows buff member



Call to attacker's address



```
int is_file_foobar( vulnerable* s, char* one,  
char* two )
```

```
{
```

```
    strcpy( s->buff, one );
```

```
    strcat( s->buff, two );
```

```
    return s->cmp( s->buff, "file://foobar" );
```

```
}
```

Heap metadata attacks

- Most heaps are implemented with metadata
 - Example: dlmalloc
- Calling `malloc` returns a buffer, potentially with extra information on each end
- Control information embedded near data
 - Attacker's dream
 - Overflow buffer, control metadata
- Complicated attack, requires in-depth understanding of malloc implementation

Format string bugs

- C stdio functions rely on format strings for identifying variable types
- `printf("%x is %s\n", addr, buf);`
 - first variable is an int, printed in hex
 - second is a NULL-terminated string
- `printf()` is variadic
- What happens if the programmer codes `printf(str)` instead of `printf("%s", str)`?

Format string bugs

- Answer: attacker can potentially read stack locations and write arbitrary memory locations
- %x characters: print stack values
 - Normally, arguments would be on the stack
- %n characters: *write* the number of characters that could be printed to the given address
- If attacker controls stack, controls write

Format string example

```
int main(int argc, char **argv)
{
    char buf[100];
    int x;
    if(argc != 2)
        exit(1);
    x = 1;
    snprintf(buf, sizeof buf, argv[1]);
    buf[sizeof buf - 1] = 0;
    return 0;
}
```

Attacker controls
format string



- See <http://seclists.org/bugtraq/2000/Sep/0214.html>

Integer overflows

- On real machines, integers have finite length
- Programmers can overflow without realizing it
- May cause unexpected bounds

```
void foo(unsigned int base,  
         unsigned int num)  
{  
    char *x = malloc(base * num);  
    // copy first element from buf  
    memcpy(x, buf, base);  
}
```

Defenses

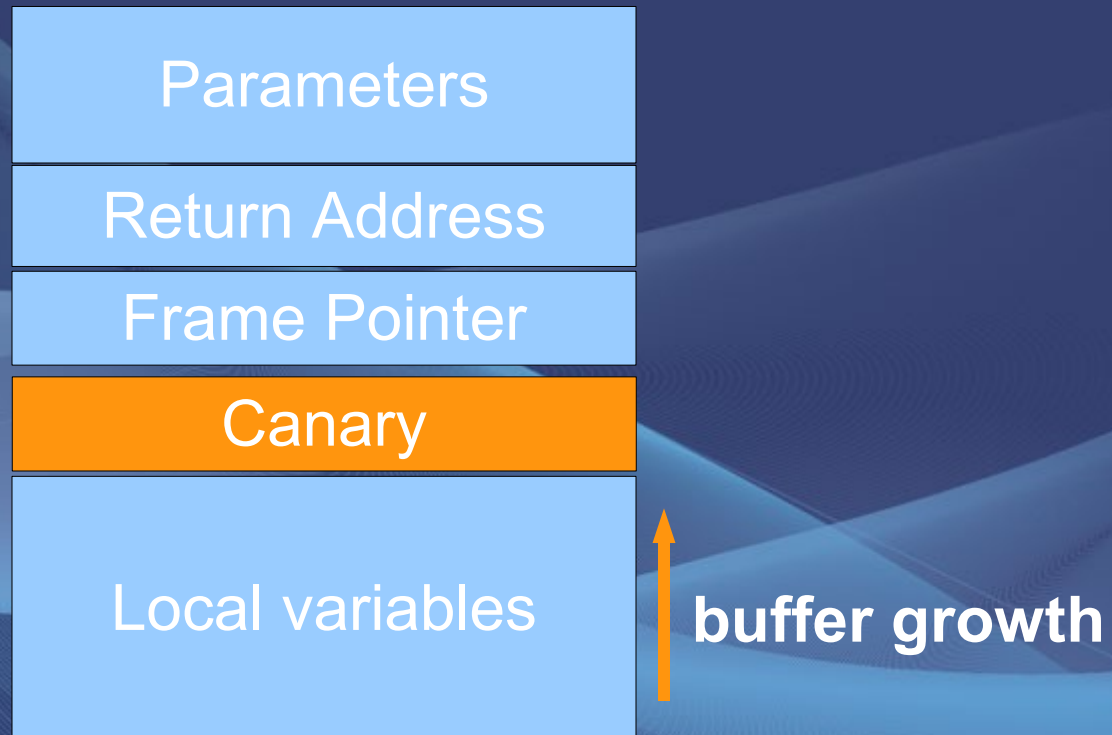
- Defenses exist at every stage
- Better requirements/threat model
- Better design
- Safer implementation
 - Better languages
 - Software analysis tools
 - Runtime mitigation strategies
- Code auditing
- Better testing

Low-level defenses

- Stack canaries
- Stack reordering
- Pointer encryption
- OS memory permission protections
- ASLR
- Execution monitoring/runtime checking

Stack canaries

- C defines return values to be abstract
- Add an extra value to the stack
- Verify value is unchanged before returning

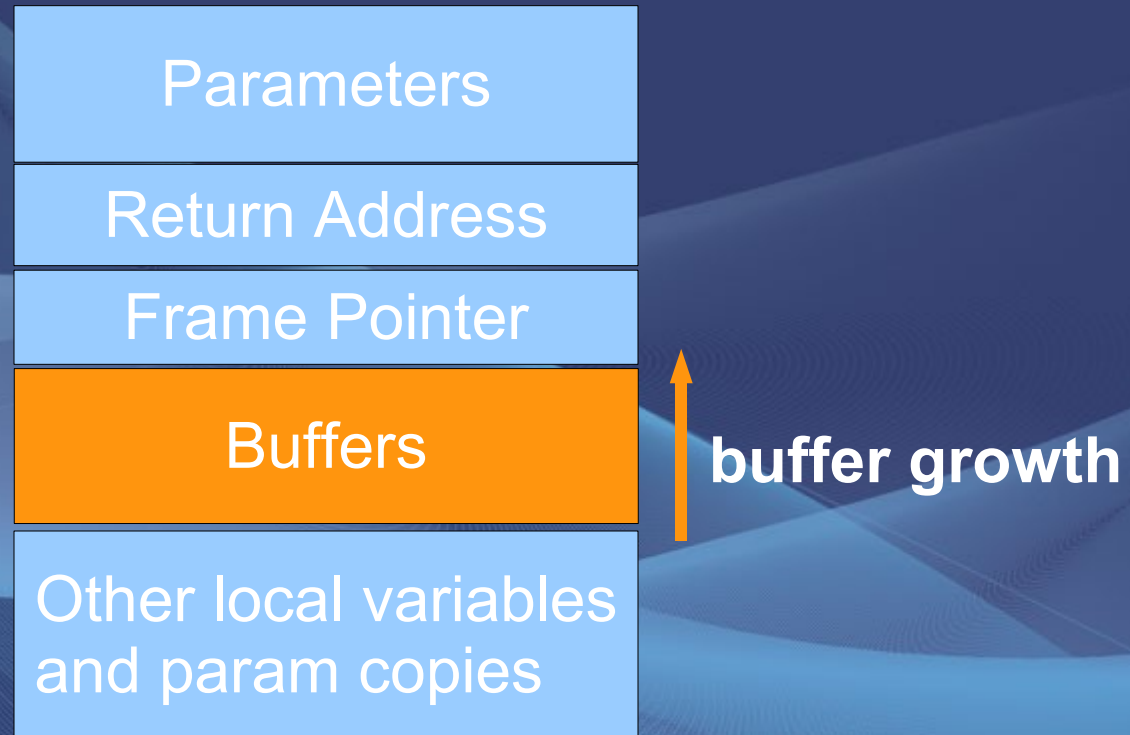


Stack canary in practice

- Choose values that attacker can't guess
- Protects against some stack smashing
 - Which won't this catch?
- Implemented in a number of modern compilers
- Small performance hit, but can use heuristics to use it less (e.g., when no buffer in frame)

Stack reordering

- Put all non-buffer arguments below buffers
- Make copies of params, put them below too
- Puts variables/params out of harms way



Stack reordering in practice

- Typically combined with canaries
- Adds little overhead
- Does not protect against all stack attacks
 - Format string in particular
- Canaries and reordering only help with stack attacks

Pointer encryption

- Represent all dynamic control changes as encrypted pointers
- Attacker needs to know key to compute a valid address
- Makes pointer arithmetic hard/expensive
- What about casts to non-pointer types?
- Used by Windows heap implementation
- Does not protect against other data attacks

W^X/Non-executable stack

- Utilize OS/hardware protections for a process to guarantee that writable data is not executable
- Available in recent x86 hardware versions
 - some software work-arounds (e.g., Linux ExecShield) for older hardware
- In most programs, stacks shouldn't be executable
- Implemented in all major operating systems (implementations differ)
 - DEP in windows
 - W^X in OpenBSD
- Only protects against code injection

ASLR

- On all major OSes, each process has its own virtual address space (modulo sharing)
- Traditionally, same program run twice has same layout
- Attackers typically need to know addresses
 - Buffer addresses
 - Code locations : return to libc
- Predictability gives attacker advantage
- Idea: randomize for harder guessing

ASLR in practice

- Implemented in many OSes
 - Linux PaX, ExecShield
 - Windows Vista
 - OSX Leopard
- Advantage over pointer encryption:
 - Mostly backwards compatible
- For complete implementation, requires compiler support (address independence)

More ASLR

- Mostly effective, but devil is in the details
 - What to do with shared libraries?
 - One answer: first time they load, get random address
 - What to do with lots of allocations?
 - True randomness would inhibit large allocations
 - 32-bit platforms have less to randomize
- Possible attacks:
 - Addresses can be “leaked”/guessed over time
 - Attacker finds indirect ways to reference things

Execution monitoring

- Assume compromise will happen
- Check for programs misbehaving
- Many systems have been devised
- Behavioral models (typically system calls)
- Program shepherding: run code in fast emulator, verify transfers before executing
- Control-flow Integrity: code only transfers according to its intended control-flow

Design Defenses

- Think security
- Risk analysis/attacker models
- Use cases
- System modeling/model checking
- Software diversity
 - Have multiple instances of same code
 - Makes it harder for attacker to cover all cases
 - ASLR is inexpensive example
 - Could have several teams develop in parallel
 - Potentially automate

Software diversity

- Harder for attacker to cover all cases
- Execution diversity
 - Same code runs differently in each instance
 - ASLR is inexpensive example
- Implementation diversity
 - Multiple instances of same design
 - Could have several teams develop in parallel
 - Potentially automate
- Functional diversity
 - Different design solutions for same system

Type Safe Languages

- Some languages make it harder to commit certain errors
- Strong type systems
- No unsafe low-level operations
- Runtime checks
- Examples
 - Java, C#, ML
 - Cyclone, CCured, Deputy
- Still, lots of legacy code out there

Language Analysis Tools

- Develop (semi)automated tools to find or reduce bugs
- Software that analyzes software
- Can run over code at any level
 - Binary, source, etc.
- Can prove general properties for *all* executions
- Undecidable for arbitrary properties
- Practical for a number of real properties
 - e.g., race detection, format string bugs

Testing/Fuzzing

- Many ways to test software
- Typically, test for expected conditions
- Fuzzing: test for *unexpected* conditions
 - Black box: look for crashes
 - White/gray box: look for abnormal behavior
- Input random/semi-structured data
- Key insight: attackers affect programs through their inputs
- Many approaches exist

Fuzzing approaches and metrics

- Mutation-based
 - Start with valid inputs, add errors
 - Requires little target knowledge, good samples
- Generation-based
 - Start with knowledge of protocol/format
 - Try to represent valid or almost valid test cases
- Common metrics
 - Code/line coverage
 - Branch coverage
 - Path coverage

Fuzzers in practice

- Used by attackers and defenders
- Many examples, some very advanced
 - Taof (The Art of Fuzzing)
 - GPF (General Purpose Fuzzer)
 - Mu-4000 (commercial)
- A number of fuzzing *frameworks* also exist
 - Give them a specification, output is fuzzer
 - e.g., SPIKE, Peach
- Hard to get good coverage
- <http://www.fuzz-test.com/presentations/>