

Lecture 18: Software Vulnerabilities

Stephen Huang

Content

1. Introduction
2. Software Vulnerabilities
3. Buffer Overflow
4. Stack Overflow

1. Introduction

- Security failures can result from malicious or non-malicious causes; they can be equally harmful.
- A security problem can arise in many places
 - Machine hardware,
 - Machine instruction,
 - Compiler,
 - Operating system,
 - Software and the user interface.
- Error, fault, failure, flaw, ...
- Benign flaws can be exploited for malicious impact.

Vulnerabilities

- Many of the vulnerabilities reported have been patched after their discoveries.
 - We are not able to demonstrate them now.
 - Some are not easily fixed.
- Nevertheless, they provide good lessons for programmers and end-users.

2. Software Vulnerabilities

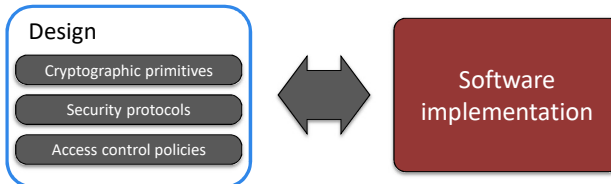


Authentication & Authorization



- Authentication: verifying the identity of the subject (e.g., user or host)
- Authorization: specifying access rights for subjects and objects

Software Vulnerabilities

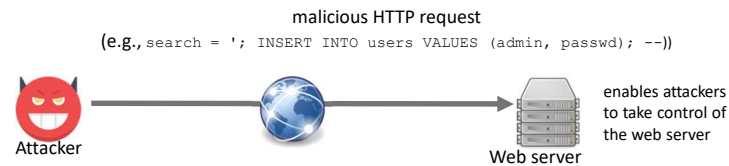


- Software vulnerability: weakness in software that allows an attacker to compromise the integrity, availability, or confidentiality of a system
- Exploiting a vulnerability can enable an attacker to
 - cause denial-of-service
 - execute arbitrary code
 - gain privileges
 - ...

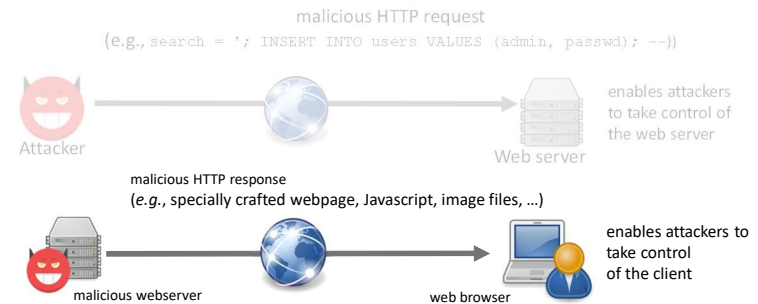
Authentication & Authorization

- Authentication is the process of verifying a user's identity, ensuring they are who they claim to be.
- Authorization, on the other hand, is the process of granting or denying access to specific resources or functionalities based on authenticated user privileges.
- Authentication confirms identity, while authorization controls access.

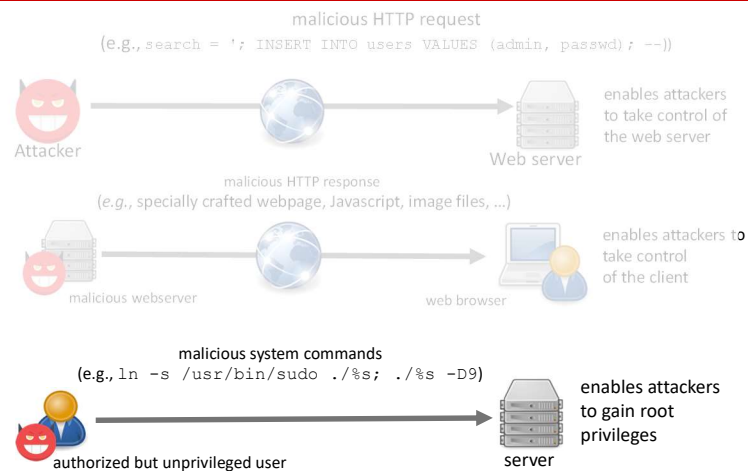
Examples of Exploiting Software Vulnerabilities



Examples of Exploiting Software Vulnerabilities

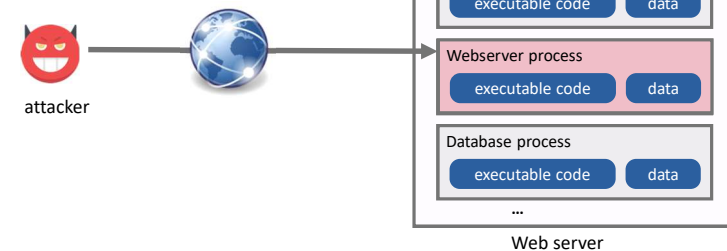


Examples of Exploiting Software Vulnerabilities

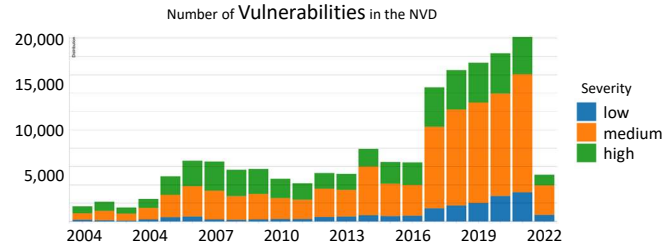


Exploitation Example

Malicious HTTP request
(e.g., search = ' ; INSERT INTO users VALUES (admin, passwd); --))



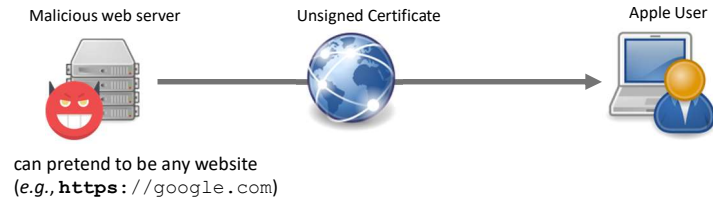
Software Vulnerability Trends in Practice



- National Vulnerability Database (NVD)
 - U.S. government repository of vulnerability management data
 - scores vulnerabilities based on attack complexity, privileges required for the attack, impact (confidentiality, integrity, availability), etc.
 - lists vulnerabilities for various widely used software (e.g., Windows, WordPress)

Apple iOS

- CVE-2014-1266:
 - The SSLVerifySignedServerKeyExchange function in ... Apple iOS ... does not check the signature in a TLS Server Key Exchange message, which allows man-in-the-middle attackers to spoof SSL servers ...



Certificate Validation

```

if (err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if (err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
if (err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
...
err = sslRawVerify(...);
...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

```

This line should not be here. The indentation is misleading.

The goto is successfully executed.

The signature checking code was skipped. Err is 0 by default which means No Error.

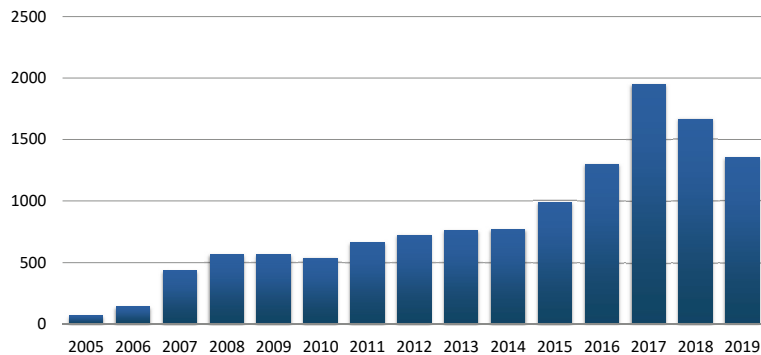
Result: invalid certificates were quietly accepted as valid.

3. Buffer Overflow

- Buffer overflow/buffer overrun:
 - anomalous condition where a process tries to store data beyond the boundaries of a fixed-length buffer
 - extra data overwrites adjacent memory, which may lead to denial-of-service or arbitrary code execution
- Attackers can supply malicious (i.e., long) input
 - remotely through a network connection (e.g., specially crafted HTTP request)
 - locally (e.g., by sending a specially crafted file to the target user)
- Vulnerable programming languages
 - C, C++, and other “lower-level” languages without bounds checking
 - “higher-level” languages, such as Java and C#, are generally not vulnerable

Buffer Overflow Vulnerability Trend

Number of Buffer Error Vulnerabilities in the NVD



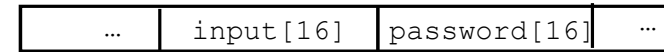
Example of Buffer Overflow

- Authenticate users by requiring them to enter a username and a correct password:

```
int authenticateUser(){
    char username[16];
    char password[16];
    char input[16];

    gets(username);
    readPasswordFromFile(username, password);
    gets(input);
    return strcmp(input, password, 16);
}
```

- Memory layout:



16 bytes

16 bytes

Example of Buffer Overflow

- Normal input (less than 16 bytes)
 - The user enters "pass123" as the password, but the correct password is "secret"
 - input does not match the password, authentication fails

Variable:	input[16]	password[16]
Contents:	pass123	secret

Example of Buffer Overflow

- Long input (more than 16 bytes)
 - example: user enters "passpasspasspasspasspasspasspass"
 - function gets does not know the length of the buffer input
 - input matches the password, authentication succeeds



Variable:	input[16]	password[16]
Contents:	pass123	secret

Variable:	input[16]	password[16]
Contents:	passpasspasspass	passpasspasspass

Typical Buffer Overflow Exploits

- Attacker's goal: execute arbitrary code (i.e., code chosen by the attacker)
- Stack-based exploitation ("stack smashing")
 - overflow a local buffer

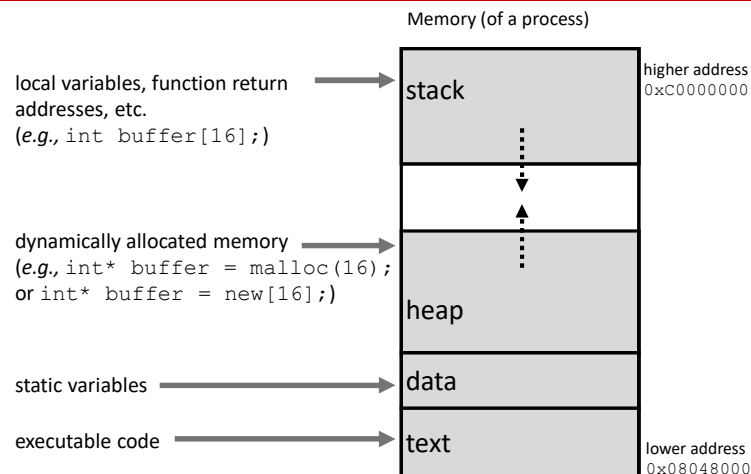

```
int buffer[16]; (see previous slide)
```
 - overwrite local variables, function return addresses, exception handlers, etc.
- Heap-based exploitation
 - overflow a dynamically allocated buffer


```
int* buffer = malloc(16 * sizeof(int));
int* buffer = new int[16];
```
 - overwrite other data, function pointers, etc.

4. Stack Overflow

- Review: Runtime environment of a program.
- A typical memory representation of a C program consists of the following sections.
 1. Text segment (i. e. instructions)
 2. Initialized data segment
 3. Uninitialized data segment (bss-block started by symbol)
 4. Heap
 5. Stack

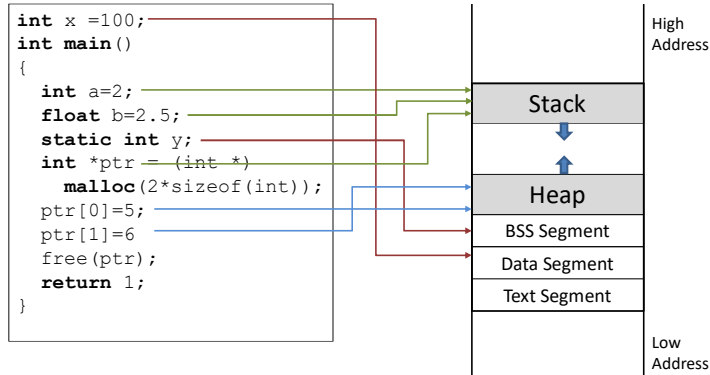
Basic Process Memory Layout



Background: Stack

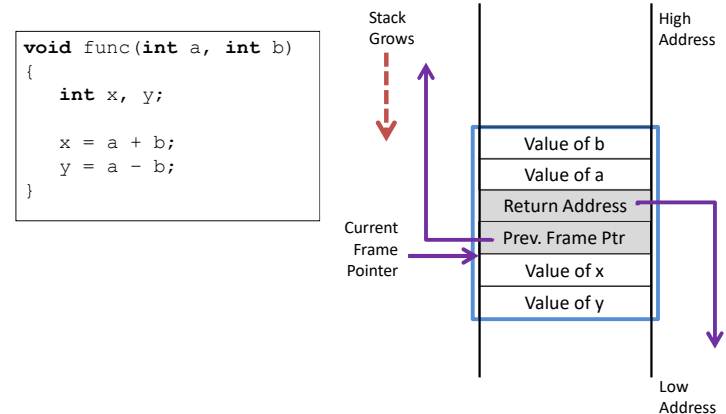
- Function calls
 - can be nested (a function may even call itself)
 - not known ahead of compile time
- Called function needs to
 - receive parameter values
 - store its local variables
 - know where to continue after it returns
- Every time a function is called, memory is "allocated" on the stack
 - stores parameters, local variables, and return address
 - frame pointer (a processor register) points to the memory "allocated" for the function
 - memory is "freed" upon return from the call

Program Memory Layout

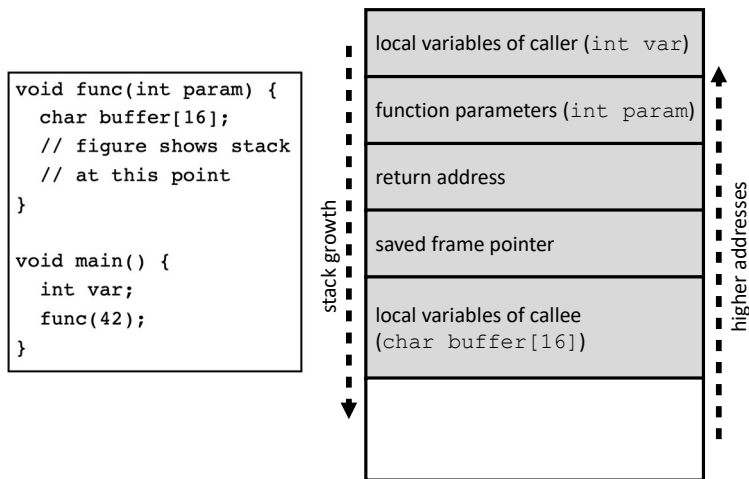


TS: Code
DS: Initialized data
BSS: block starting symbol
(uninitialized data)

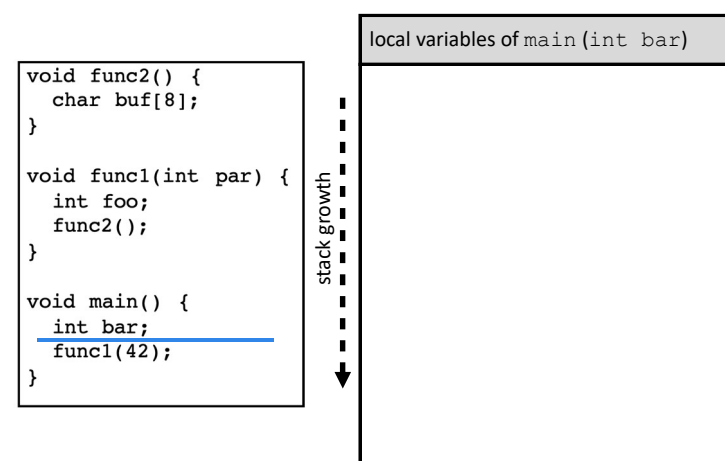
Program Memory Layout



Stack Layout



Stack Layout

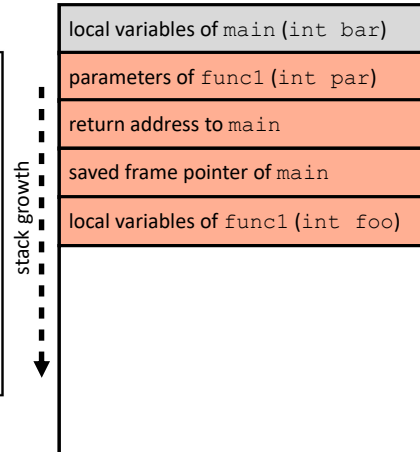


Stack Layout

```
void func2() {
    char buf[8];
}

void func1(int par) {
    int foo;
    func2();
}

void main() {
    int bar;
    func1(42);
}
```

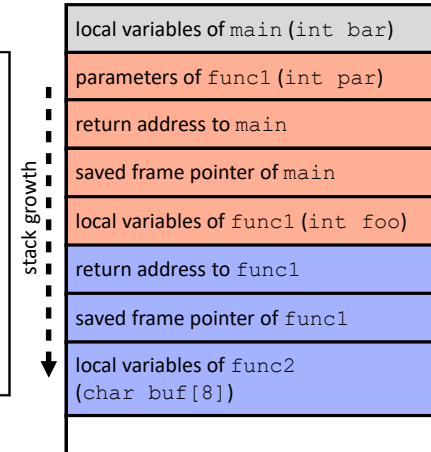


Stack Layout

```
void func2() {
    char buf[8];
}

void func1(int par) {
    int foo;
    func2();
}

void main() {
    int bar;
    func1(42);
}
```

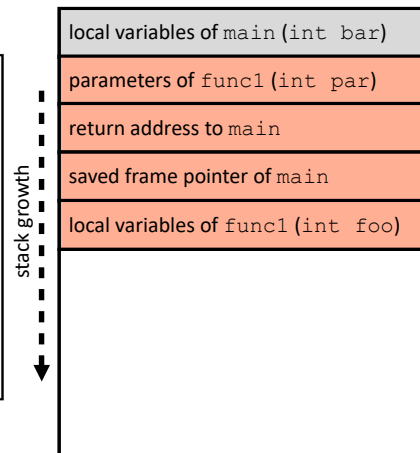


Stack Layout

```
void func2() {
    char buf[8];
}

void func1(int par) {
    int foo;
    func2();
}

void main() {
    int bar;
    func1(42);
}
```

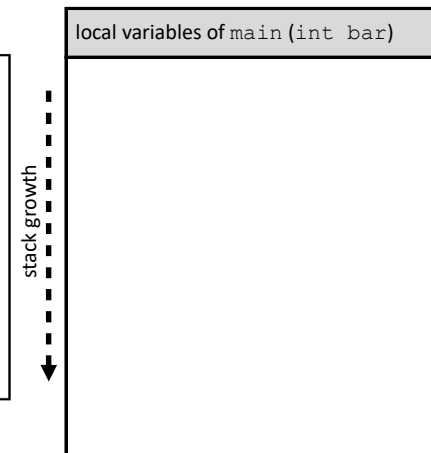


Stack Layout

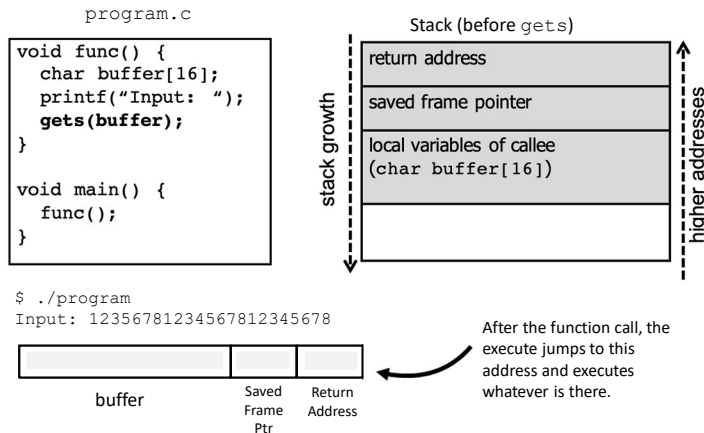
```
void func2() {
    char buf[8];
}

void func1(int par) {
    int foo;
    func2();
}

void main() {
    int bar;
    func1(42);
}
```



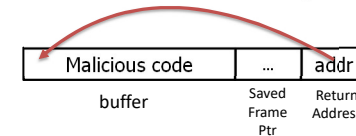
Crash



Arbitrary Code Execution

- What we can do so far:

- fill a buffer with almost any data
- change the return address of a function
- inject arbitrary malicious code into a buffer
- have the process execute the injected code



Challenges

- buffer is limited in size (duh, that is why we are able to overflow it)
- input cannot be arbitrary data (e.g., zero-terminated strings cannot contain zero value)

Shellcodes

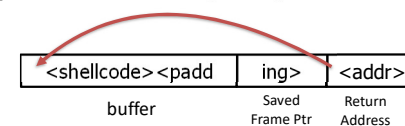
- Shellcode: small piece of code, which allows the attacker to control the compromised machine
- Example** (for 64-bit Linux)
 - taken from <http://shell-storm.org/shellcode/files/shellcode-859.php>
 - code in machine language:


```
x48 x31 xf6 x48 xf7 xe6 xff xc6 x6a x02
x5f xb0 x29 x0f x05 x52 x5e x50 x5f xb0
x32 x0f x05 xb0 x2b x0f x05 x57 x5e x48
x97 xff xce xb0 x21 x0f x05 x75 xf8 x52
x48 xbf x2f x2f x62 x69 x6e x2f x73 x68
x57 x54 x5f xb0 x3b x0f x05
```

 → only 57 bytes and no 0-valued bytes
 - starts a shell and binds it to a random network port, allowing the attacker to connect and execute arbitrary commands

Exploitation

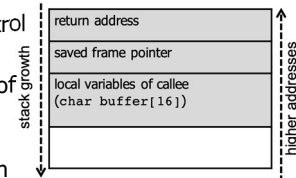
\$./program
Input: <<shellcode>><<padding>><<address>>



- When func() ends, the process starts executing the shellcode, which gives control to the attacker.
- The attacker needs to know the address of the beginning of the buffer
- The attacker may be able to debug the software on its own machine, but function calls and stack layouts can be non-deterministic.

```
void func() {
    char buffer[16];
    printf("Input: ");
    gets(buffer);
}

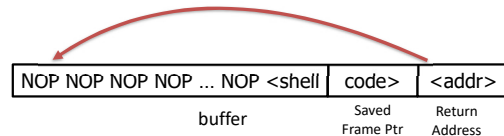
void main() {
    func();
}
```



Exploitation with NOP

```
$ ./program
```

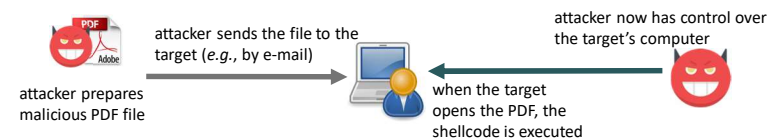
```
Input: NOP NOP ... NOP <padding><address>
```



- NOP instruction: Do nothing and move on to the next instruction.
- For successful exploitation, the return address only needs to point somewhere in the series of NOPs.

Buffer Overflow Examples: Adobe Reader

- Publicly known vulnerabilities (they have all been fixed by now)
 - CVE-2013-0610: "Stack-based buffer overflow in Adobe Reader and Acrobat 9.x before 9.5.3, 10.x before 10.1.5, and 11.x before 11.0.1 allows attackers to execute arbitrary code..."
 - CVE-2013-1376: "Buffer overflow in Adobe Reader and Acrobat 9.x before 9.5.3, 10.x before 10.1.5, and 11.x before 11.0.1 allows attackers to execute arbitrary code..."
 - CVE-2013-0626: "Stack-based buffer overflow in Adobe Reader and Acrobat 9.x before 9.5.3, 10.x before 10.1.5, and 11.x before 11.0.1 allows attackers to execute arbitrary code..."



Buffer Overflow Example: Microsoft Windows

- CVE-2008-4250
 - "The Server service in Microsoft Windows 2000 SP4, XP SP2 and SP3, Server 2003 SP1 and SP2, Vista Gold and SP1, Server 2008, and 7 Pre-Beta allows remote attackers to execute arbitrary code via a crafted RPC request that triggers the overflow during path canonicalization..."



when the server processes the request, it executes the attacker's malicious code

- Conficker worm exploited this vulnerability to spread
 - infected millions of computers in over 190 countries
 - largest known computer worm since the 2003 Welchia

Example with Source Code

- CVE-2002-0423
 - "Buffer overflow in efingerd 1.5 and earlier, and possibly up to 1.61, allows remote attackers to cause a denial of service and possibly execute arbitrary code via a finger request from an IP address with a long hostname that is obtained via a reverse DNS lookup."
- Finger protocol
 - simple protocol for exchanging human-oriented status and user information
 - implemented by efingerd on many Linux and Unix systems
- Vulnerable code:


```
static char *lookup_addr(struct in_addrin) {
    static char addr[100];
    struct hostent *he;

    he = gethostbyaddr(...);
    strcpy(addr, he->h_name);
    return addr;
}
```

Heap Overflow

- Heap overflow: overflow in data sections other than the stack
 - dynamically allocated memory
(*e.g.*, `int* buffer = malloc(16);` or `int* buffer = new[16];`)
 - statically-allocated variables (*e.g.*, global variables) initialized to zero bits
- May overwrite other data allocated on the heap or statically-allocated variables
- Examples of exploitation techniques
 - overwriting function pointers
 - *e.g.*, C++ keeps track of virtual functions by storing function pointers
 - overwriting arbitrary data by exploiting memory management
 - *e.g.*, freeing a chunk of memory on the heap links the preceding and following memory chunks together:
`follow->prec = prec` and `prec->follow = follow`

Buffer Overflow Common Culprits

- Unsafe C library functions

```
strcpy(char *dest, const char *src)
strcat(char *dest, const char *src)
gets(char *s)
scanf(const char *format, ...)
sprintf(char *s, const char *format, ...)
```


...

- Off-by-one

- *example:*

```
func f(char *input) {
    char buf[LEN];
    if (strlen(input) <= LEN)
        strcpy(buf, input);
}
```

Incorrect because one more byte is needed for the zero terminator



```
#define BUFSIZE 256
int main(int argc, char **argv) {
    char *buf;
    buf = (char *)malloc(sizeof(char)*BUFSIZE);
    strcpy(buf, argv[1]);
}
```

- ???

Heap Buffer Overflow Example

```
#define BUFSIZE 256
int main(int argc, char **argv) {
    char *buf;
    buf = (char *)malloc(sizeof(char)*BUFSIZE);
    strcpy(buf, argv[1]);
}
```

- The buffer is allocated heap memory with a fixed size, but there is no guarantee the string in `argv[1]` will not exceed this size and cause an overflow.

Privileged Instruction

- Here are some key differences between privileged and non-privileged instructions:
 - Access to resources: Privileged instructions have direct access to system resources, while non-privileged instructions have limited access.
 - Execution mode: Privileged instructions are executed in kernel mode, while non-privileged instructions are executed in user mode.
 - Execution permissions: Privileged instructions require special permissions to execute, while non-privileged instructions do not.
 - Purpose: Privileged instructions are typically used for performing low-level system operations, while non-privileged instructions are used for general-purpose computing.
 - Risks: Because privileged instructions have access to system resources, they pose a higher risk of causing system crashes or security vulnerabilities if not used carefully. Non-privileged instructions are less risky in this regard.
- <https://www.geeksforgeeks.org/privileged-and-non-privileged-instructions-in-operating-system/>

Overflow Countermeasures

- Check lengths before writing.
- Confirm that array subscripts are within limits.
- Double-check boundary condition code to catch possible off-by-one errors.
- Monitor input and accept only as many characters as can be handled.
- Use string utilities that transfer only a bounded amount of data.
- Check procedures that might overrun their space.
- Limit programs' privileges, so if a piece of code is overtaken maliciously, the violator does not acquire elevated system privileges as part of the compromise.

UNIVERSITYof **HOUSTON**

49