CSE331 Computer Security Fundamentals

9/14/2017

Program Security

Michalis Polychronakis

Stony Brook University

Software Vulnerabilities

Program flaws can turn into exploitable vulnerabilities

Securing user-space applications is equally critical as securing the OS

- May run with superuser privileges: system daemons, setuid programs, anything launched by the root account, …
- Non-privileged applications may be a stepping stone to full system compromise ➔ privilege escalation attacks
- The OS is software too

Full system compromise may not even be needed (!)

- User data is handled by user applications
- Compromising an application may be just enough
- Browsers, password managers, messaging apps, …

Compilation and Linking

Modular design is indispensable for complex applications

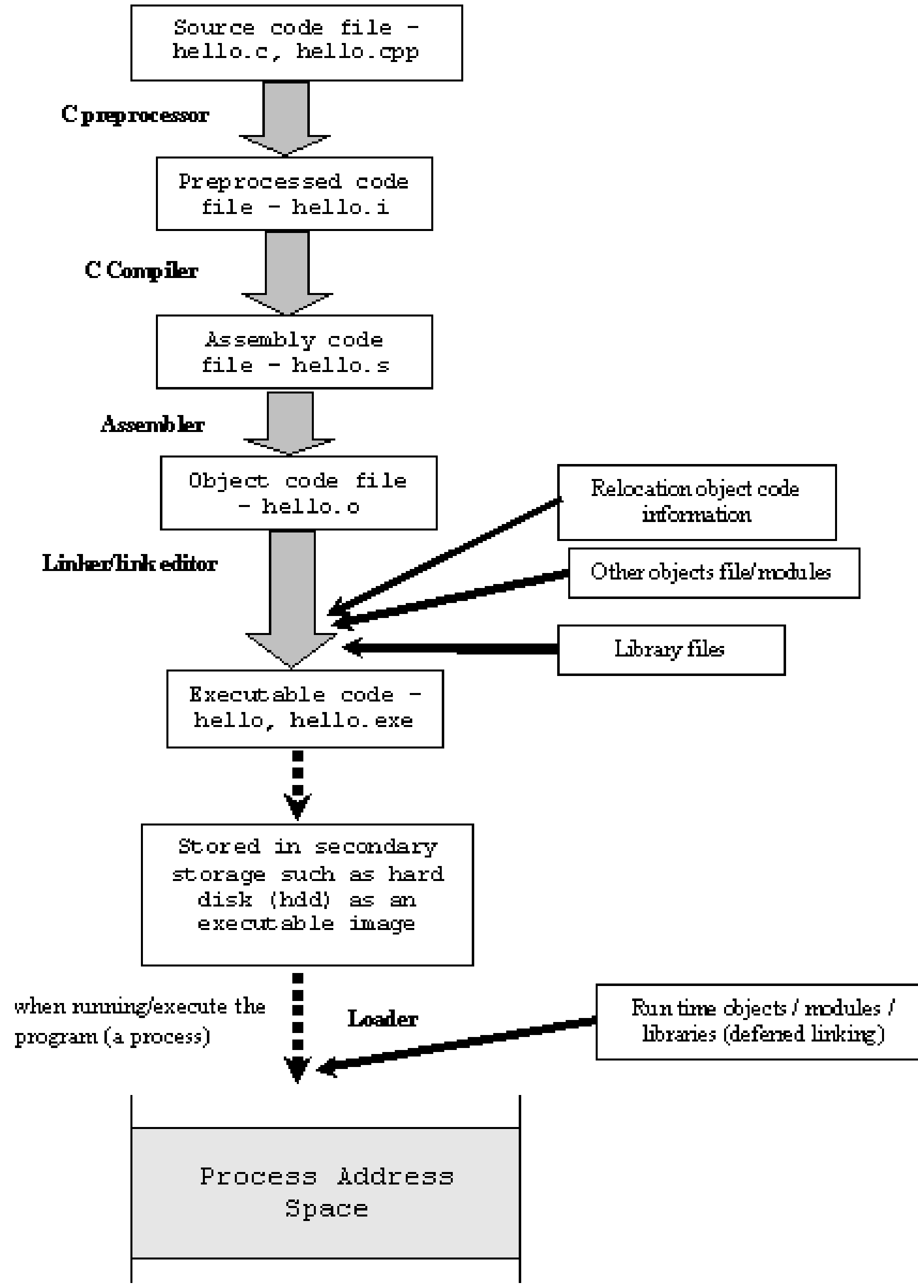
- Multiple source code files and modules

Static linking

- All libraries and other components are compiled together into a single executable

Dynamic linking

- Shared libraries are loaded separately when the program is invoked



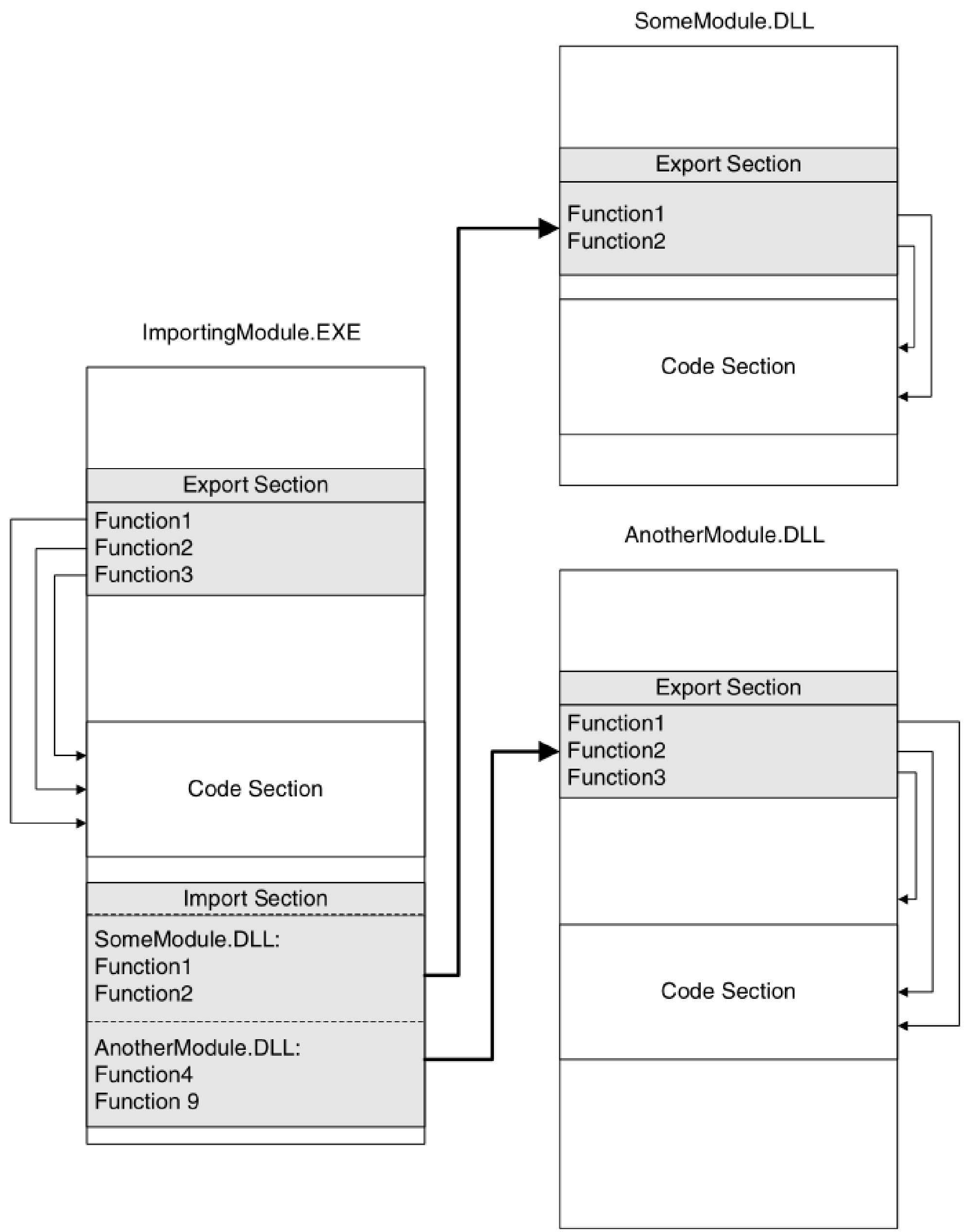
Dynamic Linking

The compiler and linker cannot know the addresses of imported functions

The linker creates an import table with all the used functions from external modules

The loader initializes the import table after modules are mapped into their final memory locations

Function addresses are found by going over the exporting module's export table

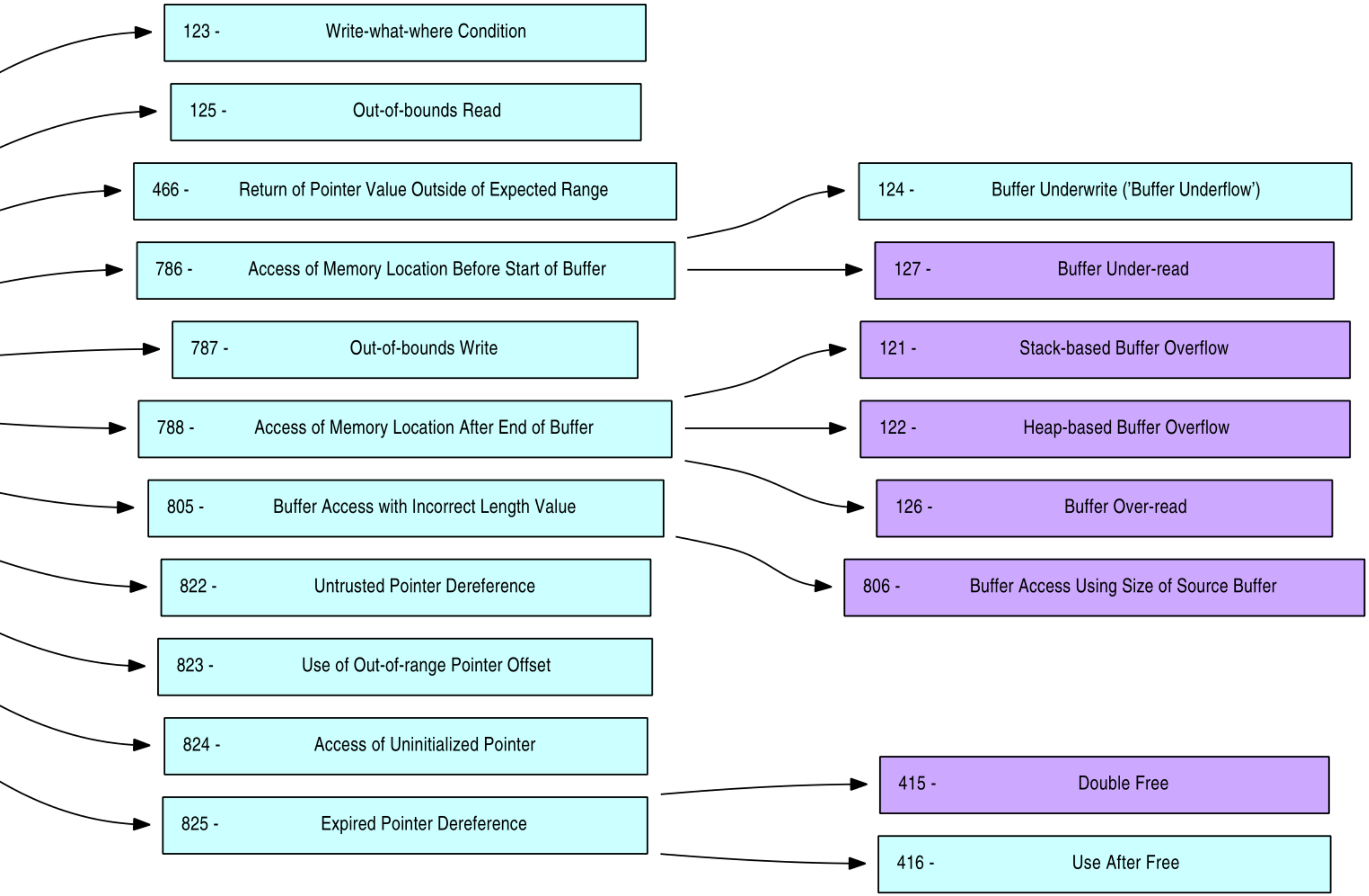


Types of Software Vulnerabilities

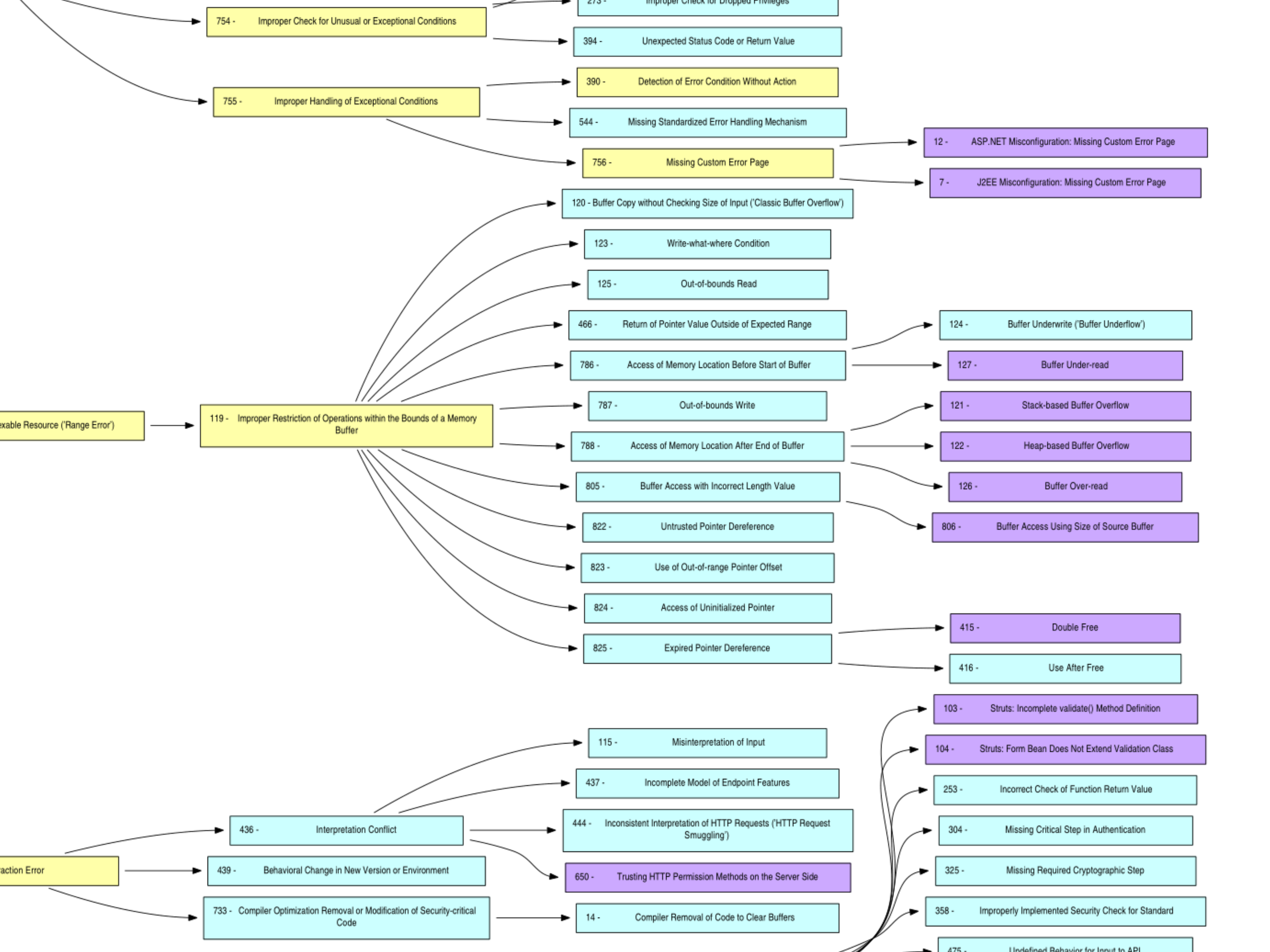
Vast number of different types of programming flaws, weaknesses, and other oversights

- Many different corresponding exploitation techniques
- Various classifications according to: type of bug, exploitation strategy, SDL phase, programming language, system layer, …

Example: MITRE's Common Weakness Enumeration (CWE) classification



123 - Write-what-where Condition
125 - Out-of-bounds Read
466 - Return of Pointer Value Outside of Expected Range
786 - Access of Memory Location Before Start of Buffer
787 - Out-of-bounds Write
788 - Access of Memory Location After End of Buffer
805 - Buffer Access with Incorrect Length Value
822 - Untrusted Pointer Dereference
823 - Use of Out-of-range Pointer Offset
824 - Access of Uninitialized Pointer
825 - Expired Pointer Dereference
124 - Buffer Underwrite ('Buffer Underflow')
127 - Buffer Under-read
121 - Stack-based Buffer Overflow
122 - Heap-based Buffer Overflow
126 - Buffer Over-read
806 - Buffer Access Using Size of Source Buffer
415 - Double Free
416 - Use After Free



754 - Improper Check for Unusual or Exceptional Conditions
394 - Unexpected Status Code or Return Value
390 - Detection of Error Condition Without Action
755 - Improper Handling of Exceptional Conditions
544 - Missing Standardized Error Handling Mechanism
12 - ASP.NET Misconfiguration: Missing Custom Error Page
756 - Missing Custom Error Page
7 - J2EE Misconfiguration: Missing Custom Error Page
120 - Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
123 - Write-what-where Condition
125 - Out-of-bounds Read
466 - Return of Pointer Value Outside of Expected Range
124 - Buffer Underwrite ('Buffer Underflow')
786 - Access of Memory Location Before Start of Buffer
127 - Buffer Under-read
787 - Out-of-bounds Write
121 - Stack-based Buffer Overflow
119 - Improper Restriction of Operations within the Bounds of a Memory Buffer
788 - Access of Memory Location After End of Buffer
122 - Heap-based Buffer Overflow
805 - Buffer Access with Incorrect Length Value
126 - Buffer Over-read
822 - Untrusted Pointer Dereference
806 - Buffer Access Using Size of Source Buffer
823 - Use of Out-of-range Pointer Offset
824 - Access of Uninitialized Pointer
415 - Double Free
825 - Expired Pointer Dereference
416 - Use After Free
103 - Struts: Incomplete validate() Method Definition
115 - Misinterpretation of Input
104 - Struts: Form Bean Does Not Extend Validation Class
437 - Incomplete Model of Endpoint Features
253 - Incorrect Check of Function Return Value
444 - Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')
436 - Interpretation Conflict
304 - Missing Critical Step in Authentication
439 - Behavioral Change in New Version or Environment
650 - Trusting HTTP Permission Methods on the Server Side
325 - Missing Required Cryptographic Step
733 - Compiler Optimization Removal or Modification of Security-critical Code
14 - Compiler Removal of Code to Clear Buffers
358 - Improperly Implemented Security Check for Standard

Another example: OWASP Top 10 (2017 rc1)

"The ten most critical web application security risks"

A1 – Injection

A2 – Broken authentication and session management

A3 – Cross-site scripting (XSS)

A4 – Broken access control

A5 – Security misconfiguration

A6 – Sensitive data exposure

A7 – Insufficient attack protection

A8 – Cross-site request forgery (CSRF)

A9 – Using components with known vulnerabilities

A10 – Unprotected APIs

Some Basic Types of Software Vulnerabilities

Memory corruption: stack/heap buffer overflow, dangling pointers, …

Arithmetic errors: arithmetic overflow, signedness, array indexing, …

Race conditions: synchronization issues, TOCTTOU bugs, …

Unvalidated input: format strings, SQL injection, command injection, …

Confused deputy: CSRF, clickjacking, …

Side channels: timing, power, temperature, …

Program logic/design/protocol flaws

Memory-related Errors

Very broad class of memory-related vulnerabilities

- One of the most important and widely exploited

In contrast to *memory safe* languages, C and C++ do not safeguard memory against illegal accesses

- Under unexpected conditions, attackers may be able to read from or write to arbitrary memory locations
- Lower-level languages ➔ performance

Operating systems, core services, desktop applications, embedded systems, and many other programs are still written in C/C++

Arithmetic Overflow

Finite number of bits to represent integers

Let's assume a 32-bit system

- Integers are expressed in *two's complement* notation

Signed integers

- Positive numbers: 0x00000000 – 0x7fffffff (0 to 2^{31}-1)
- Negative numbers: 0x80000000 – 0xffffffff (-(2^{31}) to -1)

Unsigned integers

- 0x00000000 – 0xffffffff (0 to 2^{32}-1)

Both can *overflow* or *underflow*

“Only the first 5 clients can connect”

```c
unsigned int connections = 0;
...
/* new connection attempt */
...
connections++;
if (connections < 5) {
  grant_access();
}
else {
  deny_access();
}
```

How can an attacker connect even if there are already 5 established connections?

“Only the first 5 clients can connect”

```
unsigned int connections = 0;
...
/* new connection attempt */
...
if (connections < 5) {
  connections++;
}
if (connections < 5) {
  grant_access();
}
else {
  deny_access();
}
```

Upper bound of 5 connections is enforced

Buffer Overflow

C does not provide any automatic bounds checking capability for allocated chunks of memory

- Arrays: can be indexed past the last item
- Pointers: can point outside the allocated object

Care must be taken when writing user-supplied or user-derived data into memory

- More data than expected may be supplied ➔ overflow
- The program should perform explicit bounds checks

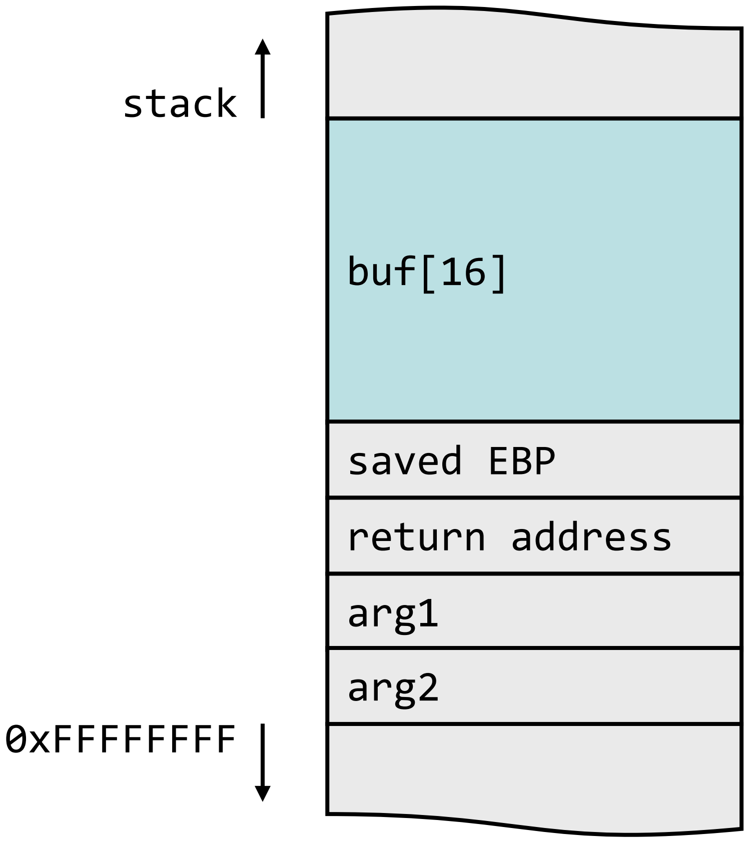
An attacker can intentionally overflow the buffer and access out-of-bounds memory

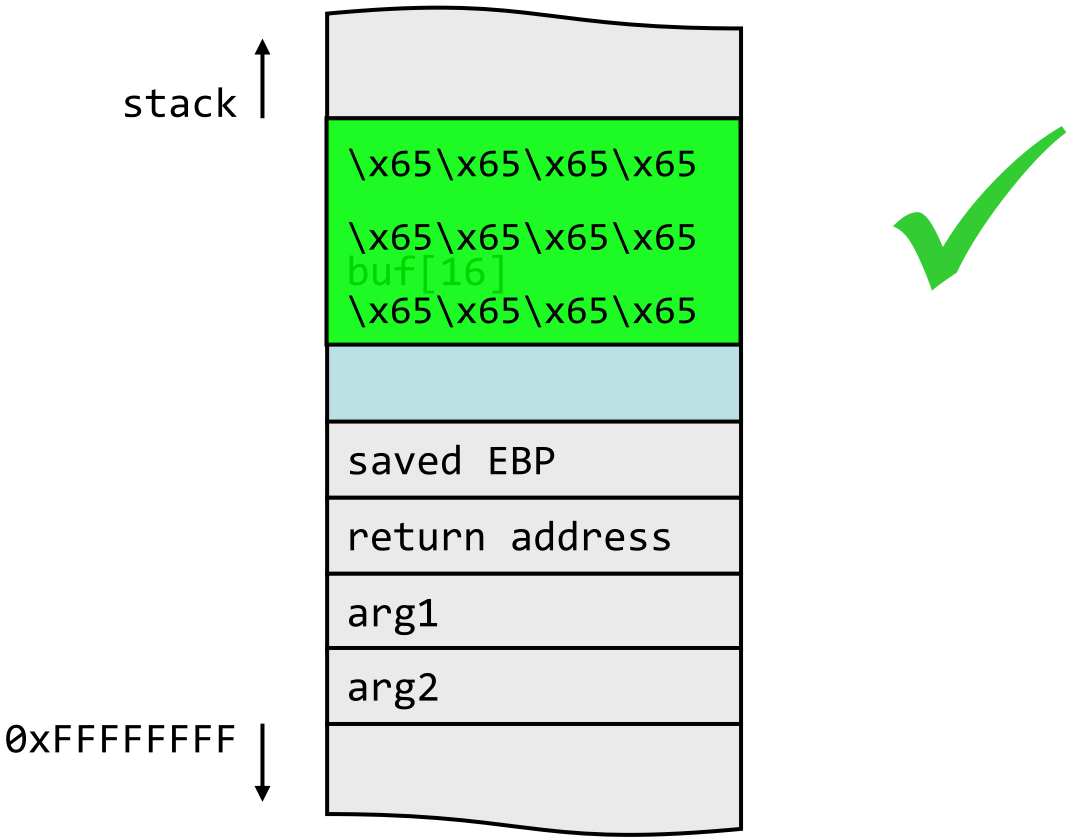
- Modify critical control or program data (overwrite)
- Leak sensitive information (overread)

Simple overflow example: unbounded string copy

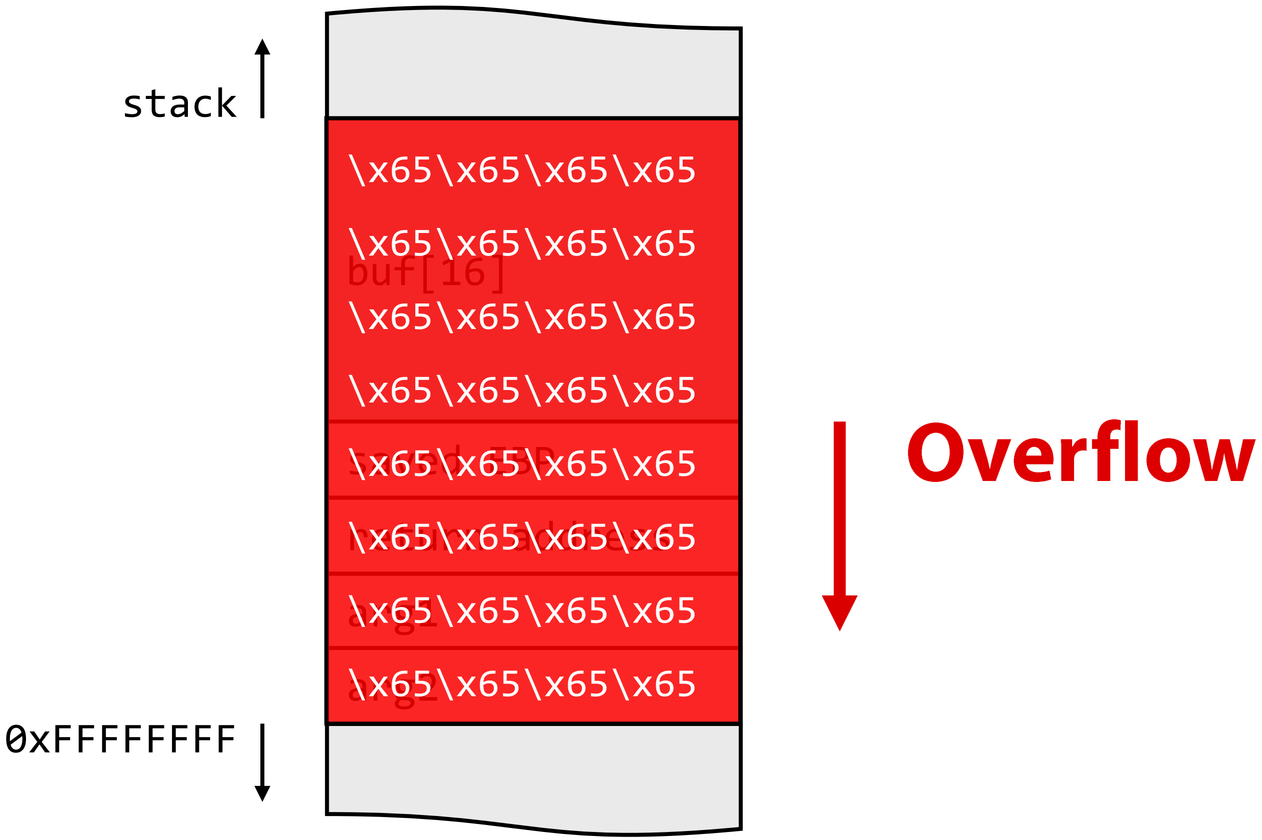
```
int main(int argc, char *argv[]) {
  char buf[16];
  strcpy(buf, argv[1]);
  printf("%s\n", buf);
  return 0;
}

$ ./overflow AAAAAAAAAAAA
AAAAAAAAAAAAAAAA
$ ./overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```


stack
buf[16]
saved EBP
return address
arg1
arg2
0xFFFFFFFF



stack
\x65\x65\x65\x65
\x65\x65\x65\x65
buf[16]
\x65\x65\x65\x65
saved EBP
return address
arg1
arg2
0xFFFFFFFF

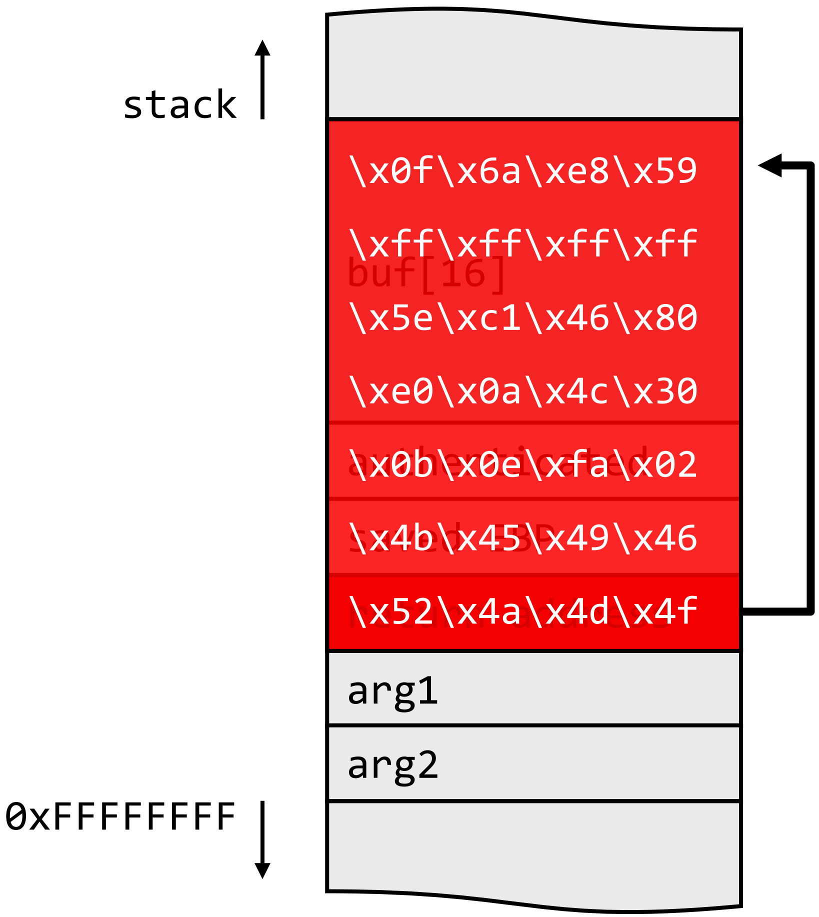

stack
\x65\x65\x65\x65
\x65\x65\x65\x65
buf[16]
\x65\x65\x65\x65
\x65\x65\x65\x65
\x65\x65\x65\x65
\x65\x65\x65\x65
\x65\x65\x65\x65
\x65\x65\x65\x65
0xFFFFFFFF
Overflow

Safer way

```
#define BUF_SIZE 16

int main(int argc, char *argv[]) {
  char buf[BUF_SIZE];
  strncpy(buf, argv[1], BUF_SIZE);
  printf("%s\n", buf);
  return 0;
}
```

What can the attacker do? ***Overwrite control data***



Shellcode injection

spawn shell

listen for connections

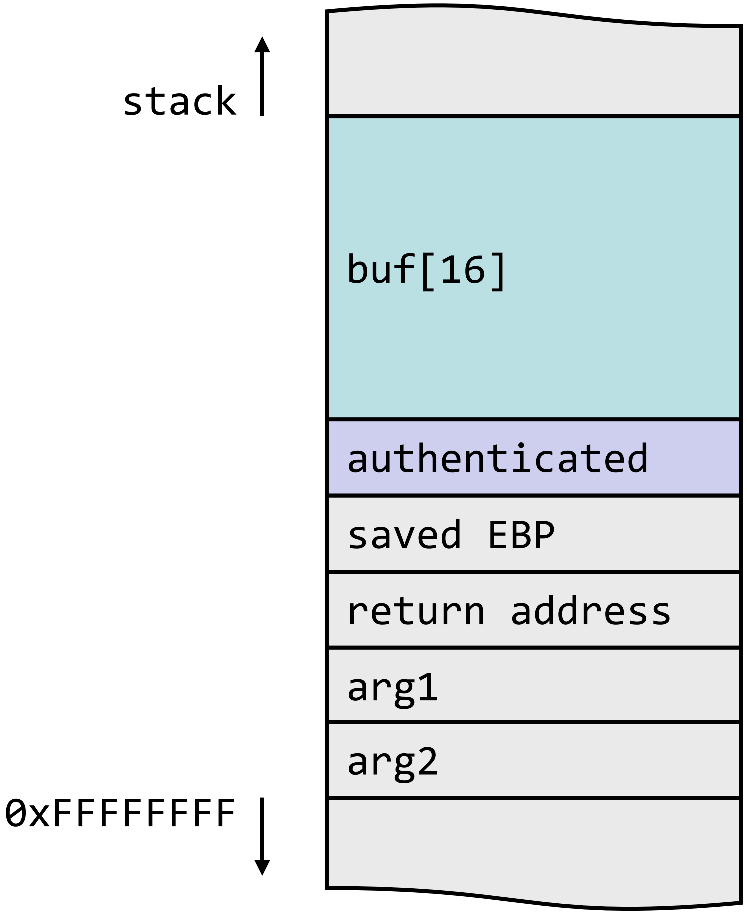
add user account

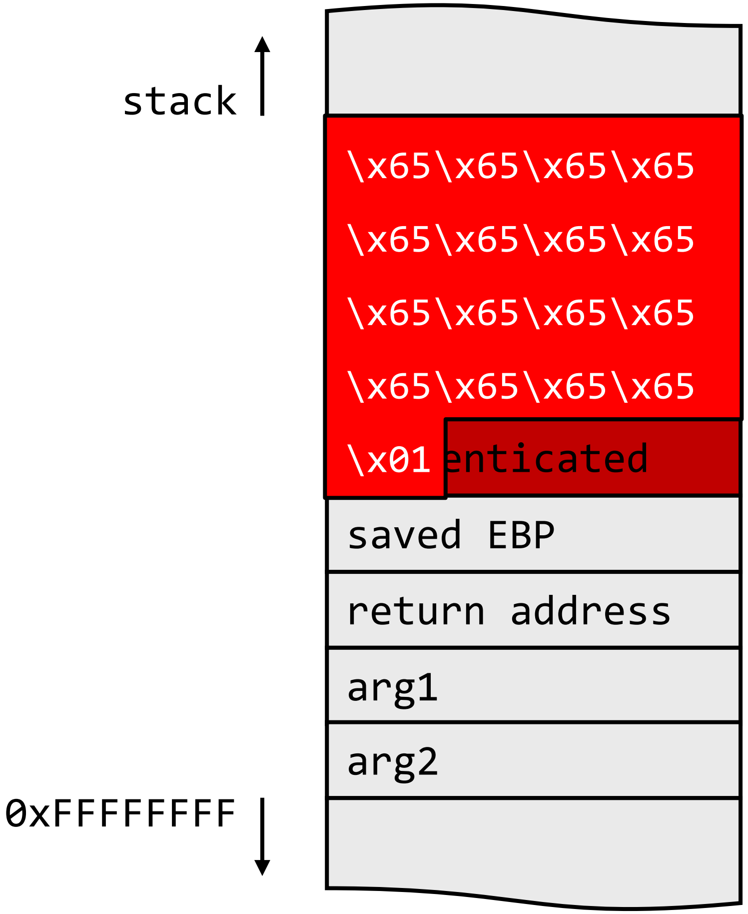
download and execute malware

(next lecture)

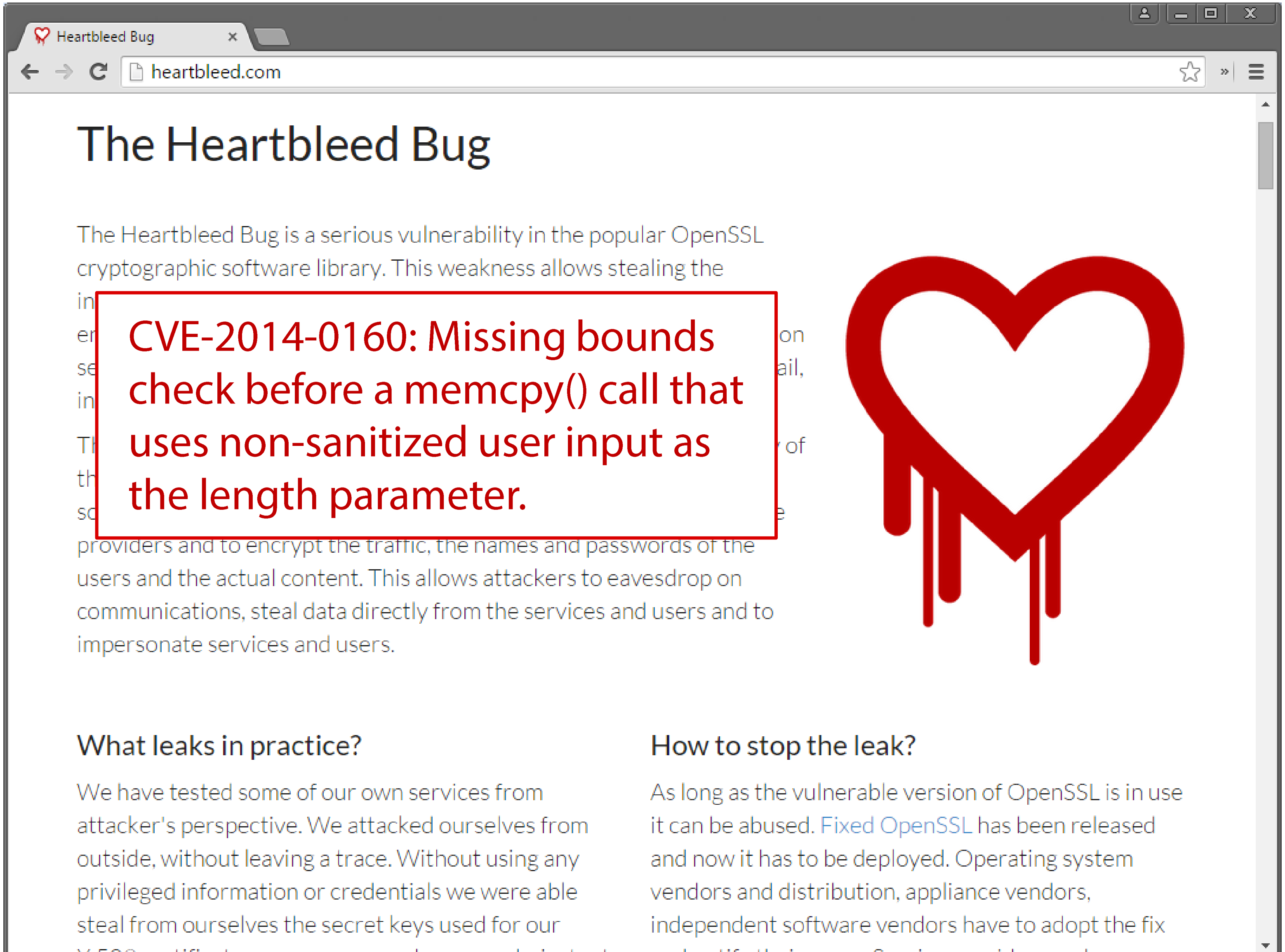
What can the attacker do? ***Overwrite program data***

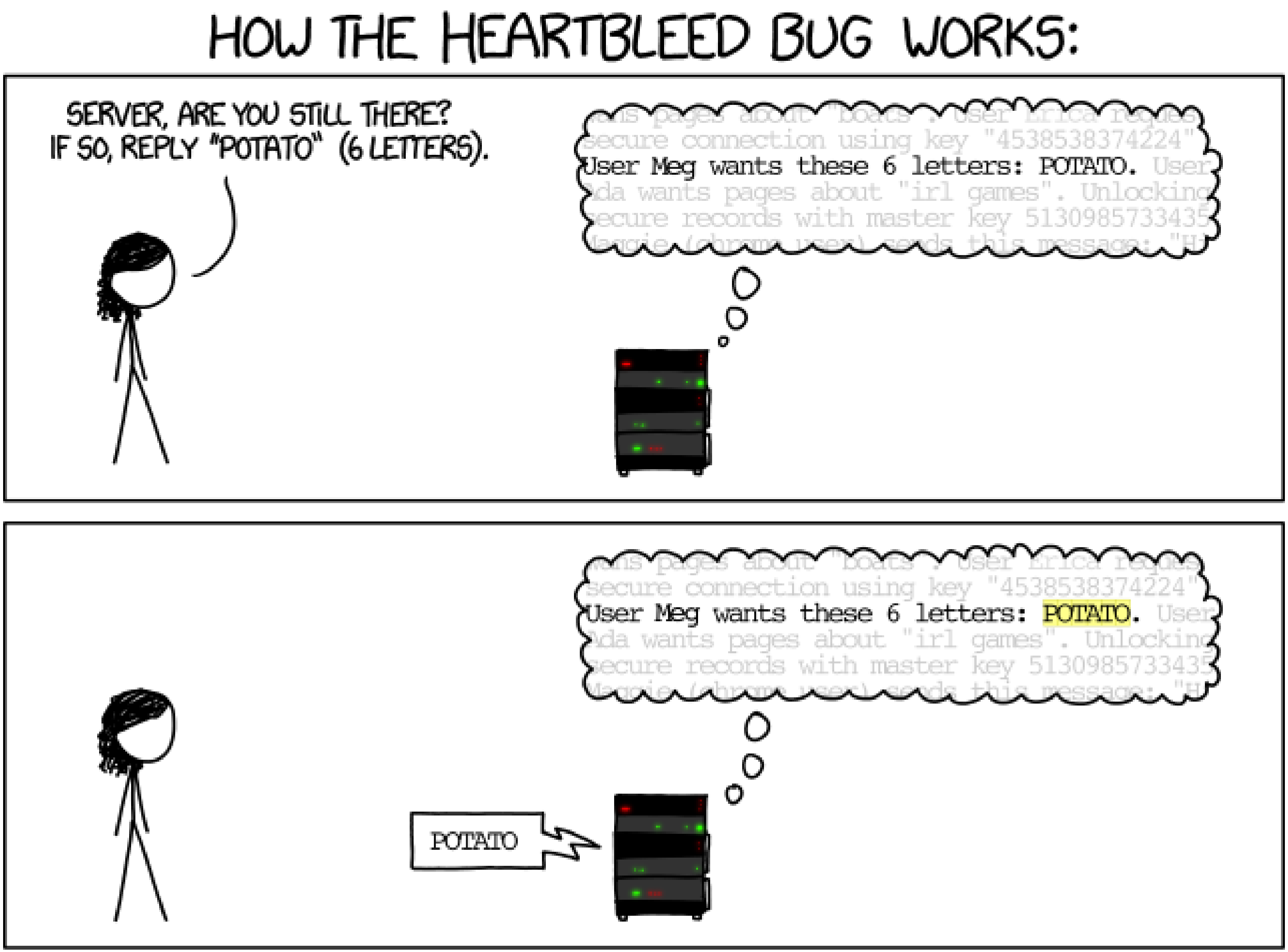
```
int main(int argc, char *argv[]) {
  int authenticated = 0;
  char password[16];
  gets(password);
  if (check_password(password) == TRUE) {
    authenticated = 1;
  }
  return authenticated;
}
$ ./authenticate AAAAAAAAAAAAAAAA && echo $?
0
$ ./authenticate AAAAAAAAAAAAAAAAA && echo $?
65
$ ./authenticate `printf "AAAAAAAAAAAAAAAA\x01"` && echo $?
1
```

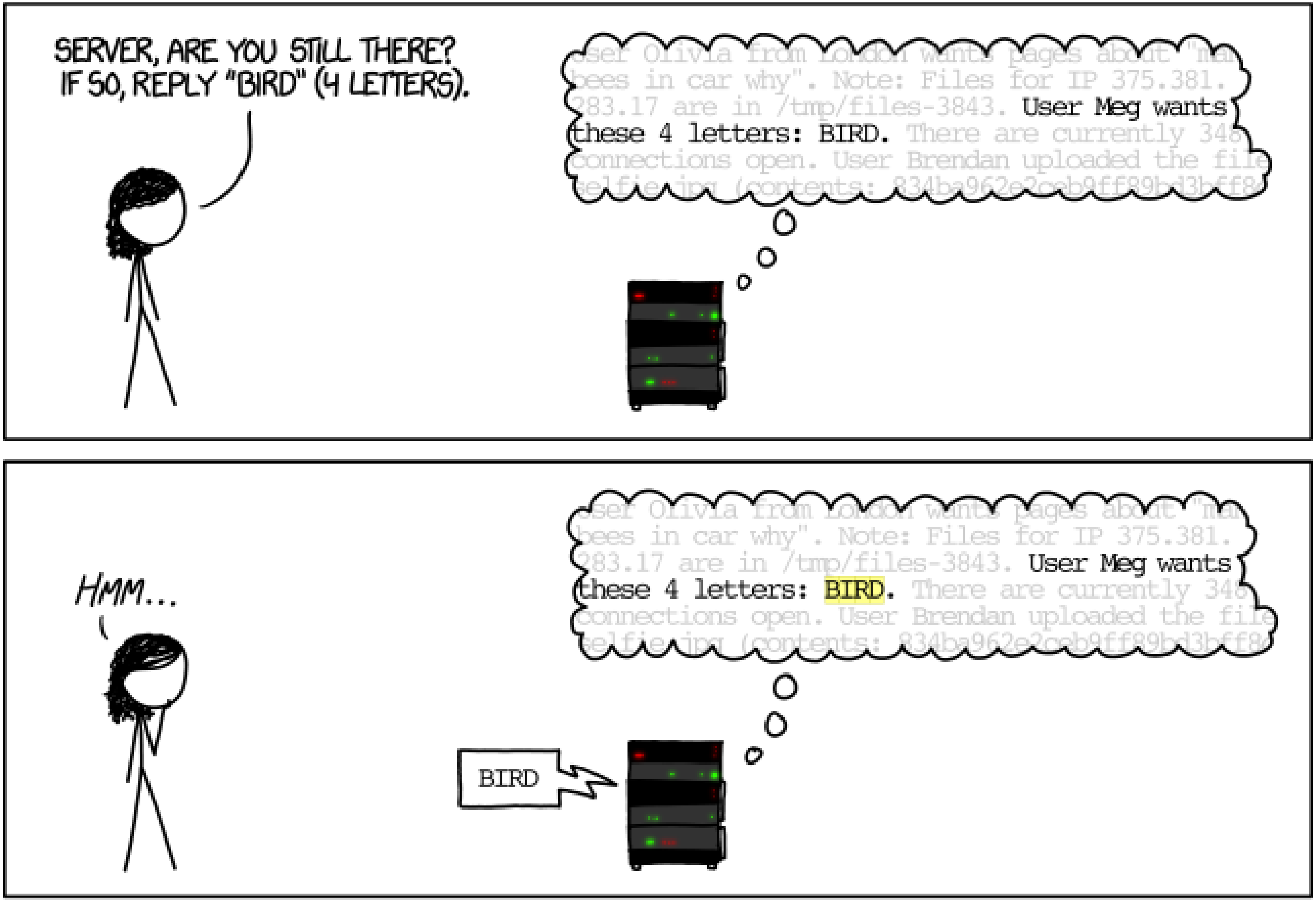

stack
buf[16]
authenticated
saved EBP
return address
arg1
arg2
0xFFFFFFFF

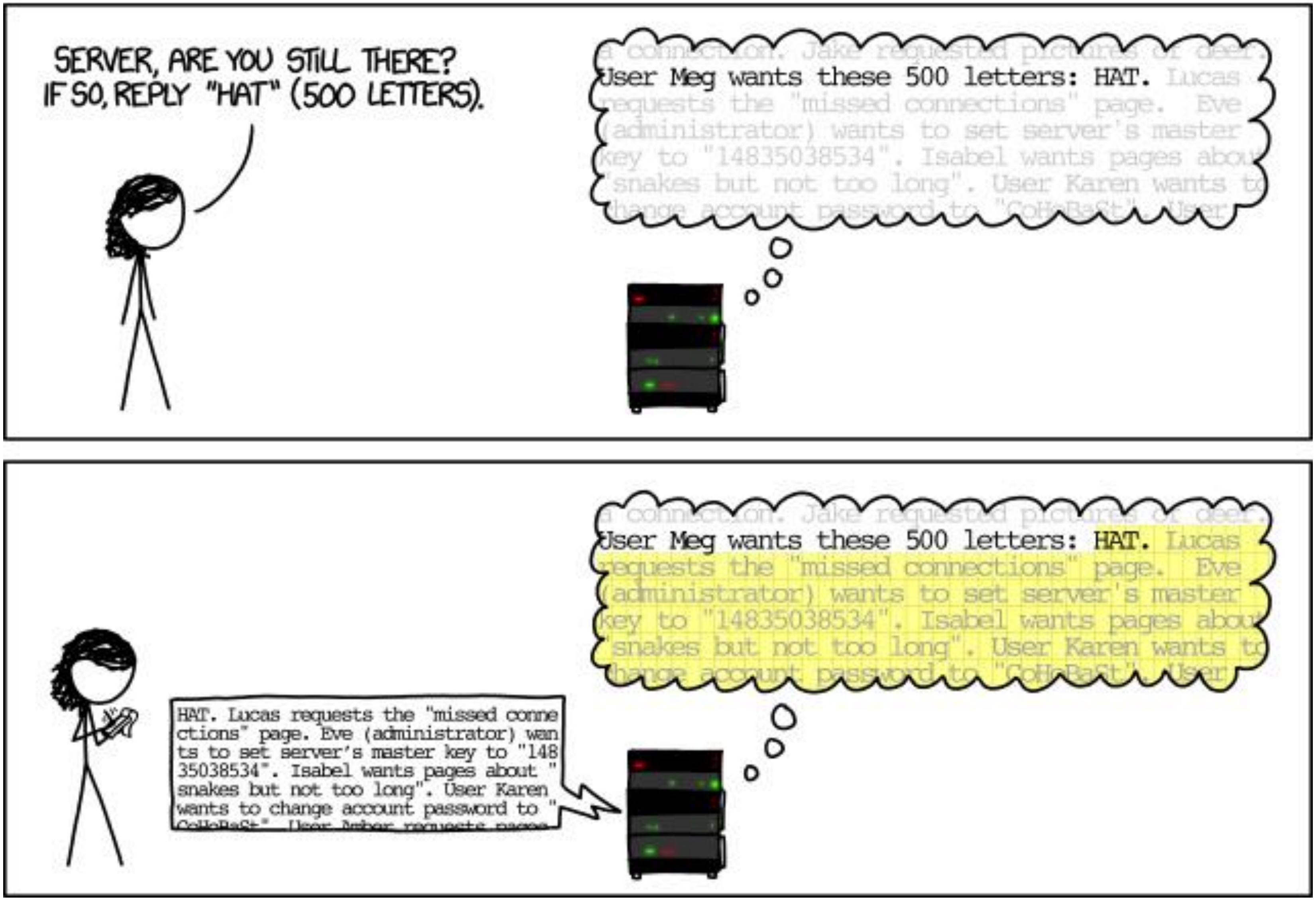

stack
\x65\x65\x65\x65
\x65\x65\x65\x65
\x65\x65\x65\x65
\x65\x65\x65\x65
\x01
enticated
saved EBP
return address
arg1
arg2
0xFFFFFFFF

What can the attacker do? ***Leak data***




HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
secure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO. User
wants pages about "irl games". Unlocking
secure records with master key 5130985733435
secure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO. User
wants pages about "irl games". Unlocking
secure records with master key 5130985733435
POTATO


SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Olivia from London wants pages about "man bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
HMM...
User Olivia from London wants pages about "man bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
BIRD


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer.
User Meg wants these 500 letters: HAT. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
snakes but not too long". User Karen wants to
change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed conne
ctions" page. Eve (administrator) wan
ts to set server's master key to "148
35038534". Isabel wants pages about "
snakes but not too long". User Karen
wants to change account password to "

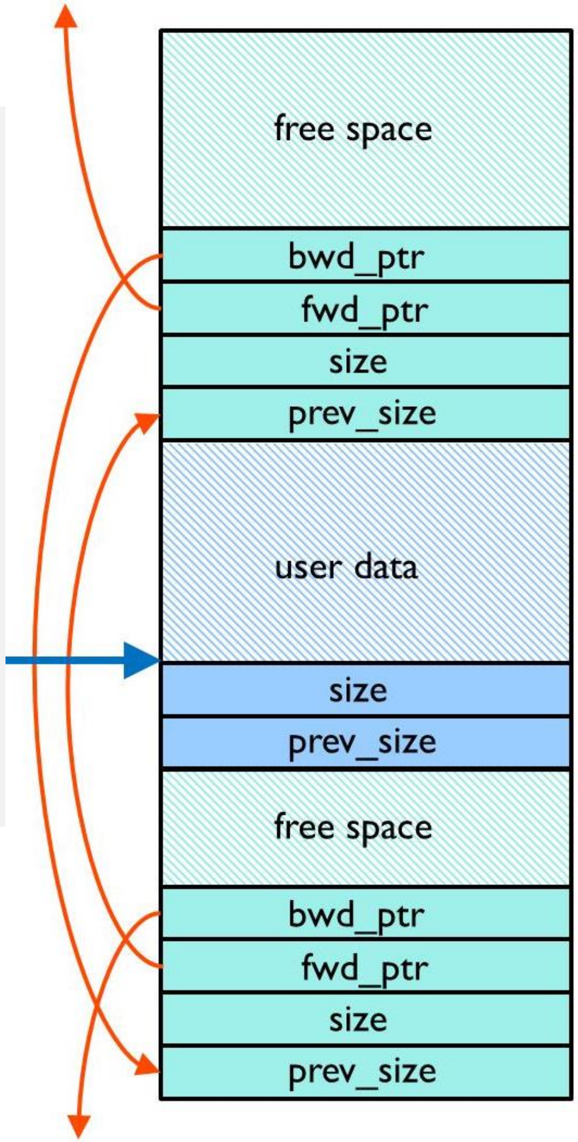
Pointer manipulation

```
void vulnerable(void *arg, size_t len) {
  long val = 0;
  long *ptr = NULL;
  char buf[128];
  ...
  memcpy(buf, arg, len);
  *ptr = val;
  ...
}
```

"Arbitrary write" capability: The attacker can write controlled data into a controlled location

Heap-based Overflows

```
int main(int argc, char *argv[]) {
  char *p, *q;
  p = malloc(1024);
  q = malloc(1024);
  strcpy(p, argv[1]);
  free(q);
  free(p);
  return 0;
}
```



Arbitrary write when `free()` is called by carefully corrupting heap metadata

Format String Vulnerabilities

The `printf()` family of functions accept a format string denoting how a variable will be displayed

- `printf("%s", str)` ➔ prints str variable as string
- `printf("%d", num)` ➔ prints num as a decimal value
- `printf("%x", num)` ➔ prints num as a hexadecimal value

Format strings can also *write* to memory

- `printf("ABCD%n", &i)` ➔ write the number of bytes output so far to the memory address of the first argument

What if…

- The programmer does not supply a format string?
- Fewer arguments are passed than the number of format string parameters?

Simple format string error example

```
int main(int argc, char *argv[]) {
  printf("Input: ");
  printf(argv[1]);
  printf("\n");
}

$ ./fmt test
Input: test

$ ./fmt "%08x %08x %08x %08x"
input: b773c080 0804846b b7721ff4 08048460

$ ./fmt $(printf "\x18\xa0\x04\x08")%x%x%x%x%n
```

Safer way

```
int main(int argc, char *argv[]) {
  printf("Input: ");
  printf("%s", argv[1]);
  printf("\n");
}
```

Other Memory-related Exploitable Conditions

NULL-termination errors

Dangling pointers

NULL pointer dereferences

String truncation

Single-byte overwrite

Off-by-one accesses

Double free

…

Race Conditions

Situations where the behavior of the program depends on the timing of some event

Critical section

- Opens up a window of opportunity for the attacker

Race conditions occur in many different contexts

- Multi-threaded programs with different threads operating on the same data
- Distributed applications that perform multi-step transactions
- Time of check to time of use (TOCTTOU): changes may happen between *checking* a condition and *using* the results of the check

Remember the Sendmail vulnerability?

Filesystem race condition example

```
// setuid program

if (access("file", W_OK) != 0) {     // access() checks the
   exit(1);                          // real uid (not eid)
}
                      ←————  ln /etc/password file

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));   // write() modifies
                                     // /etc/passwd
```

iOS 8.1 Hardware-assisted Screenlock Bruteforce

Successfully brute-force device PIN *even if "wipe out after 10 failed attempts" is enabled (!)*

Vulnerable code:

1. Display "incorrect pin" message

   ← ***Power off the device***

2. `++attempts;`

Correct code:

1. `++attempts;  // gets written to flash memory`
2. Display "incorrect pin" message


No!
Ssh!
Almost...
OK, I'm listening
Pwn3d :)
1234

Side Channels: TENEX Password Guessing Bug

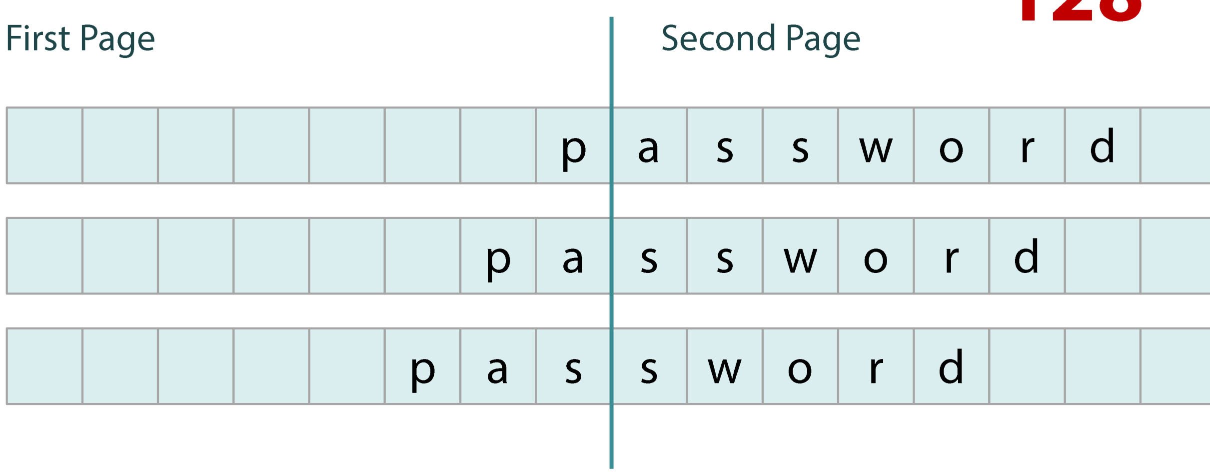
Vulnerable password checking routine

- Check each character in succession
- Report failure **on the first** mismatched character

Attack: precisely align the password buffer across two pages

- Place the first password character as the last byte of the first page
- Ensure that second page is unmapped
- Try all first characters until getting a page fault ➔ correct guess!
- Shift by one character and repeat

128^n ➔ $128n$



Program Logic Flaws: GOTO FAIL

iOS 7.0.6 signature verification error

Legitimate-looking TLS certificates with mismatched private keys were unconditionally accepted…

```
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
            goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
            goto fail;
            goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
            goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

?!!?!?!?

Check never executed