Building Security In

CMSC 330 Summer 2017
Security breaches

Just a few:

- **TJX** (2007) - 94 million records*
- **Adobe** (2013) - 150 million records, 38 million users
- **eBay** (2014) - 145 million records
- **Anthem** (2014) - Records of 80 million customers
- **Target** (2013) - 110 million records
- **Heartland** (2008) - 160 million records

*containing SSNs, credit card nums, other private info

Defects and Vulnerabilities

• Many (if not all of) these breaches begin by exploiting a **vulnerability**

• This is a *security-relevant software defect* (bug) or **design flaw** that can be **exploited** to effect an undesired behavior

• The **use of software is growing**
  • So: more bugs and flaws
  • Especially in places that are new to using software
“Internet of Things” (IOT)

Amazon Alexa

Google Home

Stuxnet specifically targets ... processes such as those used to control ... *centrifuges for separating nuclear material*. Exploiting four zero-day flaws, *Stuxnet* functions by targeting machines using the Microsoft Windows operating system ..., then seeking out Siemens Step7 software.

The result of their work was a hacking technique—what the security industry calls a zero-day exploit—that can target Jeep Cherokees and give the attacker wireless control, via the Internet, to any of thousands of vehicles.
Driverless Cars
Considering **Correctness**

- **All software is buggy**, isn’t it? Haven’t we been dealing with this for a long time?
- A **normal user never sees most bugs**, or figures out how to **work around** them
- Therefore, companies **fix the most likely bugs**, to save money
Considering **Security**

**Key difference:**

*An attacker is not a normal user!*

- The attacker will **actively attempt to find defects**, using unusual interactions and features.
  - A **typical interaction** with a bug results in a **crash**.
  - An attacker will work to **exploit** the bug to do **much worse**, to achieve his goals.
Cyber-defense?
Cyber-defense?

Popular technologies such as firewalls, anti-virus, and intrusion detection/prevention, attempt to detect the attacks themselves.

But new attacks can be produced that avoid detection but exploit the same vulnerabilities.
Penetrate and Patch

1. Find a vulnerability
2. Develop patch
3. Deploy patch (and detection signature)

But: Still vulnerable to undiscovered bugs and new bugs introduced by software upgrades.
FireEye, Kaspersky hit with zero-day flaw claims

Researchers have disclosed severe security flaws within the firm’s products over the holiday weekend.

By Charlie Osborne for Zero Day | September 8, 2015 -- 09:45 GMT (02:45 PDT) | Topic: Security

Researchers have revealed the existence of zero-day vulnerabilities within Kaspersky and FireEye’s systems which could compromise customer safety.

Over the holiday weekend, security researcher Tavis Ormandy disclosed the existence of a vulnerability which impacts on Kaspersky products. Ormandy, known in the past for publicly revealing security flaws in Sophos and ESET antivirus products, said the vulnerability is “about as bad as it gets.” In a tweet, the researcher said:

[Link to article](http://www.zdnet.com/article/fireeye-kaspersky-hit-with-zero-day-flaw-claims/)
Building Security In

The long-term solution is to prevent all exploitable bugs before deploying.

Avoid the holes to start with!
Analogy

• How do you **build a bridge** that stands up despite harsh conditions?
  • Heavy use
  • Earthquakes
  • Extreme weather
  • Etc.
Analogy

• Study the problem.

• Develop the best
• Methods
• Materials
• Tools

Then use them from Day 1!
Do not

- Use methods that **fail to incorporate larger lessons** (i.e., from past bridges built and past failures)
- **Use cheap materials** that are unresilient
- Use **unreliable tools** that produce inconsistent results
- Assume that you can do these things and everything will be OK (you can **just patch problems later**)
Unless you want your bridge to fail
From bugs to exploits
The Internet, in one slide

Client

Browser

(Private) Data

Server

Web/FTP/etc. server

Filesystem/Database/etc.

Need to **protect this state** from illicit access and tampering

(Much) user data is part of the browser

FS/DB is a separate entity, logically (and often physically)
Exploitable bugs

• Some **bugs** can be **exploited**
  • An attacker can control how the program runs so that any incorrect behavior serves the attacker

• **Many kinds of exploits** have been developed over time, with technical names like
  • Buffer overflow
  • Use after free
  • SQL injection
  • Command injection
  • Privilege escalation
  • Cross-site scripting
  • Path traversal
  • …
What is a buffer overflow?

- A buffer overflow is a dangerous bug that affects programs written in C and C++
- **Normally**, a program with this bug will simply crash
- But an **attacker** can alter the situations that cause the program to **do much worse**
  - **Steal** private information
  - **Corrupt** valuable information
  - **Run code** of the attacker’s choice
Buffer overflows from 10,000 ft

- **Buffer** =
  - Block of memory associated with a variable

- **Overflow** =
  - Put more into the buffer than it can hold

- **Where does the overflowing data go?**

*Learn more in CMSC 414!*
Buffer Overflow

```c
strcpy(buff, "abc");
```
Instructions

1. print “Password?” to the screen
2. read input into variable X
3. if X matches the password then log in
4. else print “Failed” to the screen

Data

X = abc123

Normal interaction

Password?
abc123
Failed
Instructions
1. print “Password?” to the screen
2. read input into variable X
3. if X matches the password then log in
4. else print “Failed” to the screen

Data
X = Overflow!!!!! 3.log in

Exploitation
Access granted
What happened?

- The attacker knows, for C/C++ programs
  - A buffer with the password could be a local variable
  - The typical frame layout places the caller’s return address on the stack after (higher in the address range than) local variables
  - Sometimes, the address of the stack is predictable
- Therefore
  - The attacker’s input includes machine instructions
  - The input is too long, and overruns the buffer
  - The overrun rewrites the return address to point into the buffer, at the machine code
  - When the call “returns” it executes the attacker’s code
Stopping the attack

- **Buffer overflows** rely on the ability to *read or write outside the bounds of a buffer*

- **C and C++** programs expect the *programmer* to ensure this never happens
  - But humans (regularly) make mistakes!

- Other languages (like **Python, OCaml, Java**, etc.) ensure buffer sizes are respected
  - The *compiler* inserts checks at reads/writes
  - Such checks can halt the program
  - But will **prevent a bug from being exploited**
Instructions

1. print “Password?” to the screen
2. read input into variable X
3. if X matches the password then log in
4. else print “Failed” to the screen

Data

Password?
Overflow!!!!! 3. log in

X = Overflow!

Program halted
Key idea

• The key feature of the buffer overflow attack is the attacker getting the application to treat attacker-provided data as instructions (code) or code parameters.

• This feature appears in many other exploits too:
  • SQL injection treats data as database queries
  • Cross-site scripting treats data as browser commands
  • Command injection treats data as operating system commands
  • Etc.

• Sometimes the language helps (e.g., type safety):
  • Sometimes the programmer needs to do more work.
Attack Scenarios
Interception

- Calls to remote services could be intercepted by an adversary
  - Snoop on inputs/outputs
  - Corrupt inputs/outputs

- Avoid this possibility using cryptography (CMSC 414, CMSC 456)
Malicious clients

- Server needs to **protect itself against malicious clients**
  - Won’t run the software the server expects
  - Will probe the limits of the interface
Passing the buck

- **Server needs to protect good clients** from malicious clients that will try to launch attacks via the server
  - Corrupt the server state (e.g., uploading malicious files or code)
  - Good client interaction affected as a result (e.g., getting the malware)
Defensive measures

- Two key actions the server can take:
  - **Validate that client inputs are well formed**
    - Fallacy: Focus on testing that good inputs produce good behavior
    - Must (also) ensure that malformed inputs result in benign behavior
  - Mitigate harm that might result by **minimizing the trusted computing base**
    - Isolate trusted components, or minimize privilege to precisely what is needed, in case something goes wrong
Quiz 1: What are reasonable assumptions?

Suppose you are writing a PDF viewer that is leaner and better than Acrobat Reader. Which can you assume?

A. PDF files given to your reader will always be well-formed
B. PDF files will never exceed a particular size
C. You viewer will never be used as part of an Internet-hosted service
D. None of the above
Quiz 1: What are reasonable assumptions?

Suppose you are writing a PDF viewer that is leaner and better than Acrobat Reader. Which can you assume?

A. PDF files given to your reader will always be well-formed
B. PDF files will never exceed a particular size
C. You viewer will never be used as part of an Internet-hosted service
D. None of the above
Validating inputs
What’s wrong with this Ruby code?

```ruby
if ARGV.length < 1 then
  puts "required argument: textfile path"
  exit 1
end

# call cat command on given argument
system("cat "+ARGV[0])

exit 0
```

`catwrapper.rb`
Possible Interaction

> ls
  catwrapper.rb
  hello.txt

> ruby catwrapper.rb hello.txt
Hello world!

> ruby catwrapper.rb catwrapper.rb
if ARGV.length < 1 then
  puts "required argument: textfile path"
...

> ruby catwrapper.rb "hello.txt; rm hello.txt"
Hello world!

> ls
  catwrapper.rb
Quiz 2: What happened?

A. **cat** was given a file named `hello.txt`; `rm hello.txt` which doesn’t exist

B. **system()** interpreted the string as having two commands, and executed them both

C. **cat** was given three files – `hello.txt`; and `rm` and `hello.txt` – but halted when it couldn’t find the second of these

D. **ARGV[0]** contains `hello.txt` (only), which was then catted

```ruby
# catwrapper.rb:
if ARGV.length < 1 then
  puts "required argument: textfile path"
  exit 1
end

# call cat command on given argument
system("cat "+ARGV[0])

exit 0
```

> ruby catwrapper.rb “hello.txt; rm hello.txt”

`Hello world!`

> ls

`catwrapper.rb`
Quiz 2: What happened?

A. `cat` was given a file named `hello.txt`; `rm hello.txt` which doesn’t exist

B. `system()` interpreted the string as having two commands, and executed them both

C. `cat` was given three files – `hello.txt`; and `rm` and `hello.txt` – but halted when it couldn’t find the second of these

D. `ARGV[0]` contains `hello.txt` (only), which was then catted

---

catwrapper.rb:
```ruby
if ARGV.length < 1 then
  puts "required argument: textfile path"
  exit 1
end

# call cat command on given argument
system("cat "+ARGV[0])
exit 0
```

> ruby catwrapper.rb “hello.txt; rm hello.txt”
*Hello world!*

> ls
*catwrapper.rb*
Possible deployment

Client

Browser

GET foo.txt

<output>

Server

Web server

catwrapper.rb
Consequences?

- If `catwrapper.rb` is part of a web service
  - **Input is untrusted** — could be anything
  - But we only want requestors to read (see) the contents of the files, not to do anything else
  - Current code is too powerful: vulnerable to

  **command injection**

- How to fix it?

  **Need to validate inputs**

  [https://www.owasp.org/index.php/Command_Injection](https://www.owasp.org/index.php/Command_Injection)
Input Validation

• We expect input of a certain form
  • But we cannot guarantee it always has it
    - it’s under the attacker’s control
  • So we must validate it before we trust it

• Making input trustworthy
  • Sanitize it by modifying it or using it in such a way that the result is correctly formed by construction
  • Check it has the expected form, and reject it if not
Checking: Blacklisting

- **Reject** strings with possibly bad chars: ' ; --

```ruby
if ARGV[0] =~ /;/ then
  puts "illegal argument"
  exit 1
else
  system("cat "+ARGV[0])
end
```

> ruby catwrapper.rb "hello.txt; rm hello.txt"

*illegal argument*
Sanitization: Blacklisting

• Delete the characters you don’t want: ' ; --

```ruby
system("cat "+ARGV[0].tr(";",""))
```

```bash
> ruby catwrapper.rb "hello.txt; rm hello.txt"
Hello world!
cat: rm: No such file or directory
Hello world!
> ls hello.txt
hello.txt
```
Sanitization: Escaping

- **Replace problematic characters with safe ones**
  - change ‘ to \\
  - change ; to \\
  - change – to \\
  - change \ to \\

- Which characters are problematic depends on the interpreter the string will be handed to
  - Web browser/server for URIs
    - `URI::escape(str,unsafe_chars)`
  - Program delegated to by web server
    - `CGI::escape(str)`
Sanitization: Escaping

```ruby
def escape_chars(string)
    pat = /('|"|\.|\*|\/|\-|\\;|\s)/
    string.gsub(pat){|match|"\" + match}
end

system("cat "+escape_chars(ARGV[0]))
```

> ruby catwrapper.rb "hello.txt; rm hello.txt"
cat: hello.txt; rm hello.txt: No such file or directory
> ls hello.txt
hello.txt
Quiz 3: Is this escaping sufficient?

A. No, you should also escape character &
B. No, some of the escaped characters are dangerous even when escaped
C. Both of the above
D. Yes, it’s all good

```ruby
def escape_chars(string)
  pat = /\'|"|\.|\*|\//\-\|\|\|s)/
  string.gsub(pat){|match|"\" + match}
end
system("cat "+escape_chars(ARGV[0]))
```
Quiz 3: Is this escaping sufficient?

A. No, you should also escape character &
B. No, some of the escaped characters are dangerous even when escaped
C. Both of the above
D. Yes, it’s all good

---

catwrapper.rb:

```ruby
def escape_chars(string)
  pat = /\([^\] times\s\})/; \$
  string.gsub(pat){|match|"\\" + match}
end
system("cat "+escape_chars(ARGV[0]))
```
Escaping not always enough

> ls ../passwd.txt
passwd.txt
> ruby catwrapper.rb "../passwd.txt"
bob:apassword
alice:anotherpassword

- A web service probably only wants to give access to the files in the current directory
  - the .. sequence should have been disallowed

- Previous escaping doesn’t help because . is replaced with \\ which the shell interprets as .
Path traversal

This is called a path traversal vulnerability. Solutions:

- Delete all occurrences of the . character
  - Will disallow legitimate files with dots in them (hello.txt)

- Delete occurrences of .. sequences
  - Safe, but disallows foo/../hello.txt where foo is a subdirectory in the current working directory (CWD)

- Ideally: Allow any path that is within the CWD or one of its subdirectories

https://www.owasp.org/index.php/Path_Traversal
Checking: Whitelisting

- **Check that the user input is known to be safe**
  - E.g., only those files that exactly match a filename in the current directory

- **Rationale**: Given an invalid input, **safer to reject than to fix**
  - “Fixes” may result in wrong output, or vulnerabilities
  - **Principle of fail-safe defaults**
Checking: Whitelisting

files = Dir.entries(".").reject { |f| File.directory?(f) }

if not (files.member? ARGV[0]) then
  puts "illegal argument"
  exit 1
else
  system("cat "+ARGV[0])
end

> ruby catwrapper.rb "hello.txt; rm hello.txt"
illegal argument
Validation Challenges

- **Cannot always delete or sanitize problematic characters**
  - You may want dangerous chars, e.g., “Peter O’Connor”
  - How do you know if/when the characters are bad?
  - Hard to think of all of the possible characters to eliminate

- **Cannot always identify whitelist cheaply or completely**
  - May be expensive to compute at runtime
  - May be hard to describe (e.g., “all possible proper names”)

56
Key Questions

• Which inputs in my program should not be trusted?
  • These start from input from untrusted sources
  • And these inputs influence (“taint”) other data that flows through my program
    - And could be stored in files, databases, etc.

• How to ensure that untrusted inputs, no matter what they are, will produce benign results?
  • Sanitization, checking, etc. as early as possible
    - How to do this depends on the program, and how the inputs are used
Quiz 4: As a developer, security is

A. Something I can help address by writing better code  
B. Something that writing better code can do little to address  
C. Something that is the purview of the government, e.g., DHS  
D. Something that will never be solved so long as market forces do not value security

(Pick an answer you think is best)