

CS1632: Security Testing

Wonsun Ahn

Writing Secure Software Is Difficult; So Is Testing It!

- **Heartbleed:** A defect in OpenSSL

- Caused ~ 66% of servers connected to the Internet to be vulnerable
- Allowed for untraceable eavesdropping on data in memory
- Discovered in 2014, vulnerability introduced in 2012



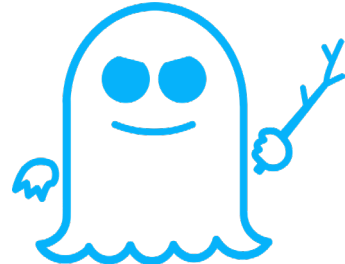
- **Shellshock:** A defect in bash (default shell for OS X and most Linux)

- Millions of attacks recorded in the days following discovery
- Allowed arbitrary code execution stored in environment variables
- Discovered in 2014, vulnerability introduced in 1989



Even Security Testing *Hardware* is Difficult

- **Spectre / Meltdown:** A vulnerability in CPU design
 - Impacts all CPUs in wide-use today (Intel, AMD, ARM, IBM ...)
 - Allows arbitrary access to private data in a process (Spectre)
 - Allows arbitrary access to private data in an OS (Meltdown)
 - Discovered in 2017, vulnerability introduced in **1995**
 - OS / Web Browser patches issued but some Spectre vulnerabilities still open



SPECTRE

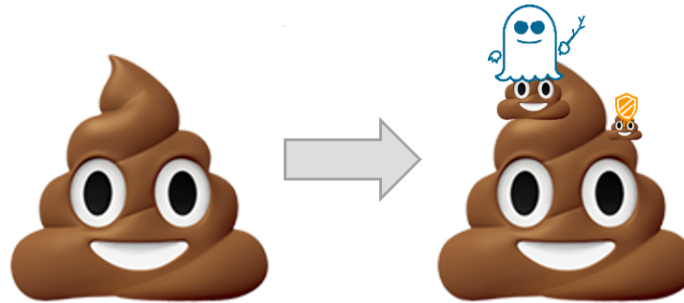


MELTDOWN

A Slide from a 2018 Hardware Design Conference

Risk in context

Because of software bugs, computer security was in a dire situation



Spectre doesn't change the magnitude of the risk, but adds to the mess

- Poor mitigation options (fixes -> new risks)

Why is it so Difficult?

1. Adversaries are actively seeking to defeat security.
2. Information about security vulnerabilities spreads quickly.
3. You need to protect all doors. They only need to open one.
4. Important open source software is not properly funded.
 - OpenSSL defect caused an estimated \$500 million in damages
 - Only one fulltime volunteer was maintaining it on \$2000 yearly donations
 - Aftermath: Core Infrastructure Initiative formed to fund open source software
<https://www.coreinfrastructure.org/>

History

- Security was not a big deal in the early computing world
 - Usually required physical access to a system to do anything
 - Few people had necessary skills even if they did have access (So called “security through obscurity”)
- Hacker culture 1960-80s exemplified in ITS Operating System
 - OS did not use passwords; anyone could use it and do anything
 - There was a flaw where clever users could crash the OS. Solution?
 - A “crash” command was created that could be run by anyone
 - Crashing the OS was not challenging or fun anymore → nobody did it

History

- Now the stakes are much higher
 - “Estimating the Global Cost of Cyber Risk”, RAND Corp., 2018
https://www.rand.org/pubs/research_reports/RR2299.html
 - Global cost of cyber crime: \$799 billion to \$22.5 trillion
(1.1% to 32.4% of global GDP)
- And there are many more actors, not all with good intentions ...

Actors in the Security Sphere

- **White hat hackers** (Ethical hackers)
 - Performs *penetration testing* (or *pen testing*) for a client to report vulnerabilities
 - Employees of a security firm or in-house security teams of organizations
- **Black hat hackers** (Crackers)
 - Violates system security for personal gain or other malicious purpose
- **Red hat hackers** (Hacktivists)
 - Violates system security to spread a political / ideological / religious message
- Organized crime (works in conjunction with black hat hackers)
- Nation states (e.g. Stuxnet, Equation Group)

The InfoSec Triad: CIA

- **InfoSec Triad** (Information Security Triad)
 - Three attributes that define a system that provides information security
 - It's summarized by the three letters: CIA
 - No, it has nothing to do with the Central Intelligence Agency
- **CIA** as in:
 - **Confidentiality** - No *unauthorized* users may *read* data
 - **Integrity** - No *unauthorized* users may *write* data
 - **Availability** - System is *available* for reading or writing
- A security attack compromises one (or more) of these attributes

Security Attack Categorization

1. Attacks on **Confidentiality: Interception**

- **Eavesdropping:** Monitoring messages on an unsecured network
- **Keylogging:** Monitoring key strokes on a computer using a surreptitious software
- **Phishing:** Forging a legit company email or website to extract info

2. Attacks on **Integrity: Modification / Fabrication**

- Malware that formats hard disk of computer, or fills it with garbage data
- **Digital signature forgery:**
 - Electronic messages are “signed” using a digital signature to prove authenticity and integrity.
 - In this attack, the message is modified or fabricated and then the signature is forged.

3. Attacks on **Availability: Interruption**

- **DoS (Denial of Service):** Sending a flood of messages to a server, shutting it down
- **Power grid attack:** Attacking power source of a server, shutting it down

Vulnerability vs Exploit vs Attack

- **Vulnerability:** Weakness of a system
- **Exploit:** Mechanism for compromising a system
- **Attack:** Actual compromising of the system (one of the CIA attributes)
- In a nutshell: An exploit *enables* an attack *using* a vulnerability. E.g.:
 - Vulnerability: Array bounds not checked when accessing a C array
 - Exploit: *Buffer overflow* to access beyond the bounds of the array
 - Attack: A compromise of one of CIA using buffer overflow
- **Malware:** Malicious code used to perform an attack

Malware Categorized by type of Attack

- **Spyware:** covertly *monitors* your actions (eavesdroppers, keyloggers, ...)
- **Adware:** *shows* you more ads when browsing to a webpage
- **Ransomware:** threatens to *publish* data or *block* access until ransom is paid
- **Bacteria:** program that *consumes* system resources (e.g. fork bomb)
- **DoS:** a program that floods a server with messages to *shut it down*

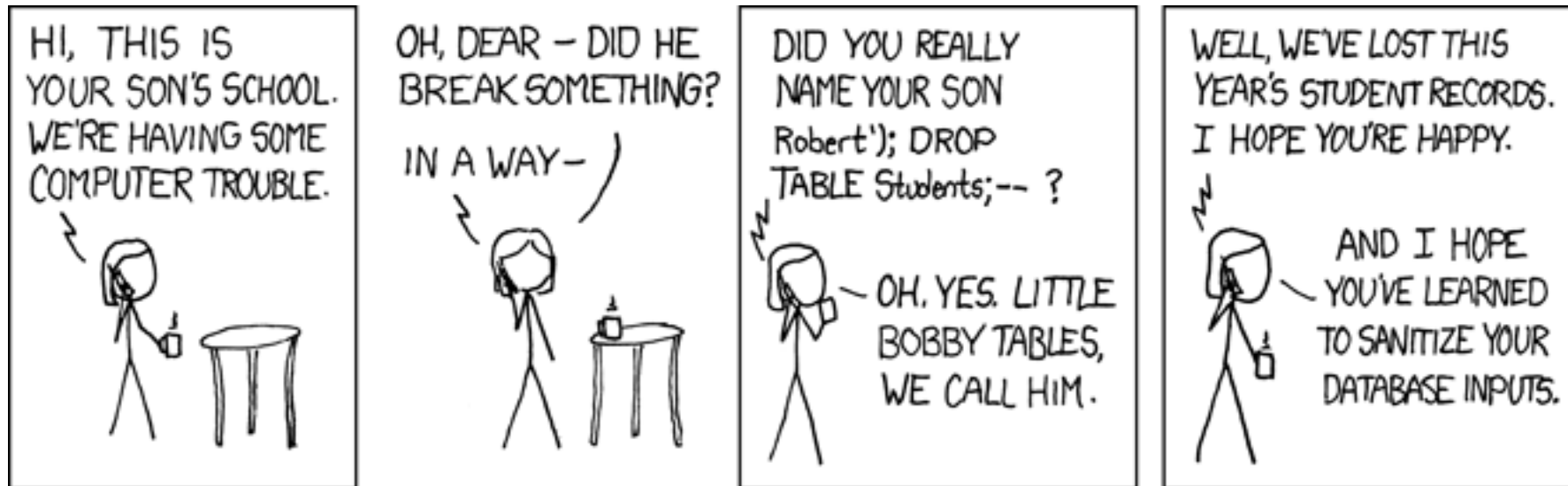
Malware Categorized by type of Exploit

- **Logic bomb** – *hidden* code within program that sets off attack when *triggered*
- **Trapdoor** - *secret* undocumented *access* to a system or app
- **Trojan horse** – program that *pretends* to be another program
- **Virus** - *replicates* itself WITH human intervention
- **Worm** - *replicates* itself WITHOUT human intervention
- **Zombie** – a computer or program being run by an *unauthorized controller*
- **Bot network** – collection of zombies controlled by master

Examples of well-known Exploits

- Injection Attacks
- Cross-Site Scripting (XSS)
- Insecure Object Reference Exploit
- Buffer overflow
- Broken Authentication Exploit
- Security Misconfiguration Exploit
- Insecure Storage Exploit
- Social Engineering

Injection Attacks



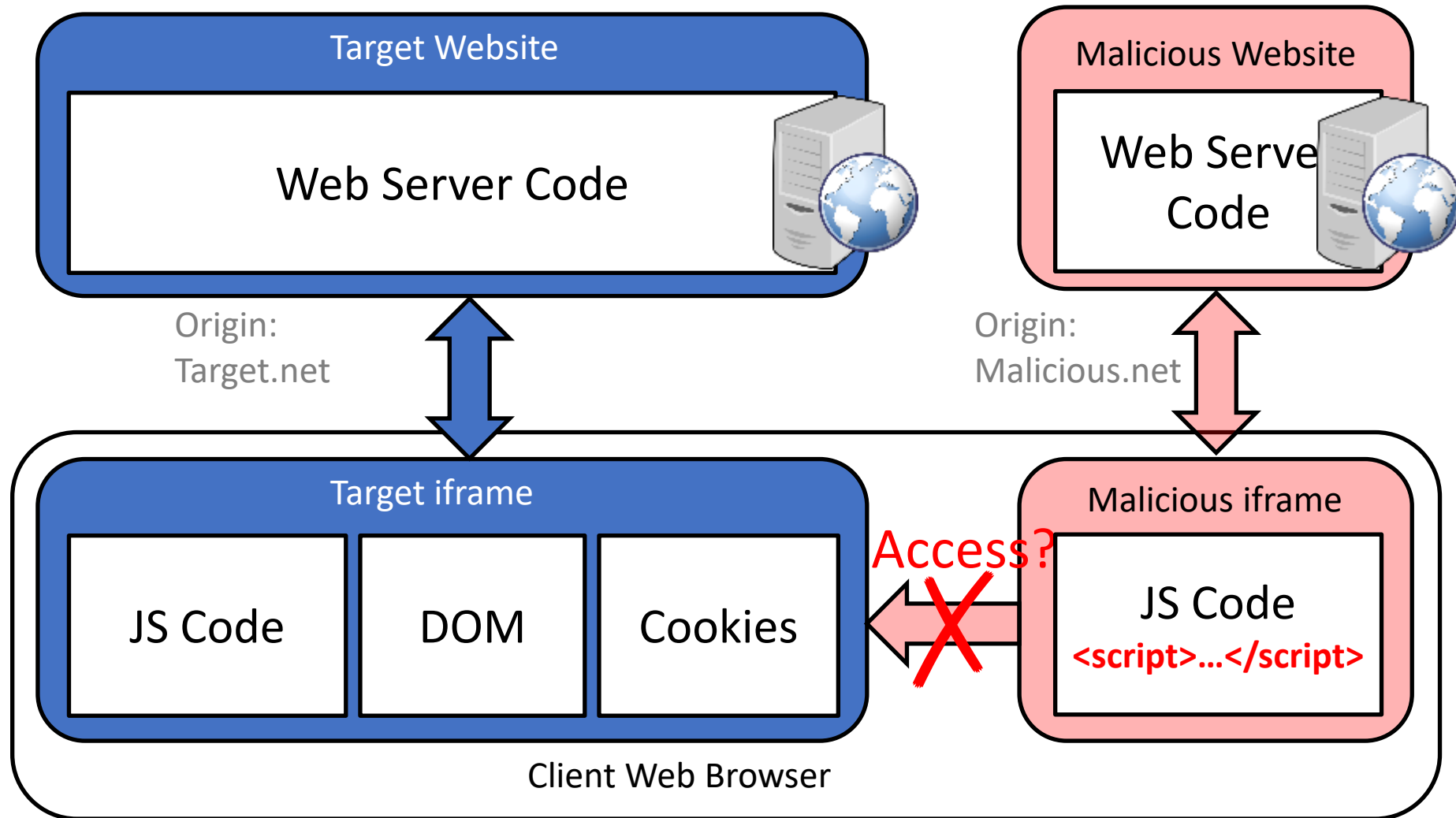
Injection Attacks: Solution

- Yes, sanitizing any user input is a good idea
- *Sanitization*: scrubbing user input to prevent it from injecting code
 - E.g. Only allowing alphabets in a name field to prevent SQL code injection
 - E.g. Not allowing `<script> ... </script>` tags to prevent JavaScript code injection
- Even better, use SQL *parameterization*
 - Telling database engine that a string should be treated as parameter not code
 - https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html

Cross-Site Scripting

- 2019 CWE (Common Weakness Enumeration) Top 25: 2nd place
 - The most popular exploit for web apps for over a decade
- To fully understand, need to first understand Same Origin Policy
- Same Origin Policy (SOP): Web browser sandboxing architecture
 - A frame can access data in another frame only if from same URL origin
 - E.g. An advertisement frame cannot access data in your online banking page

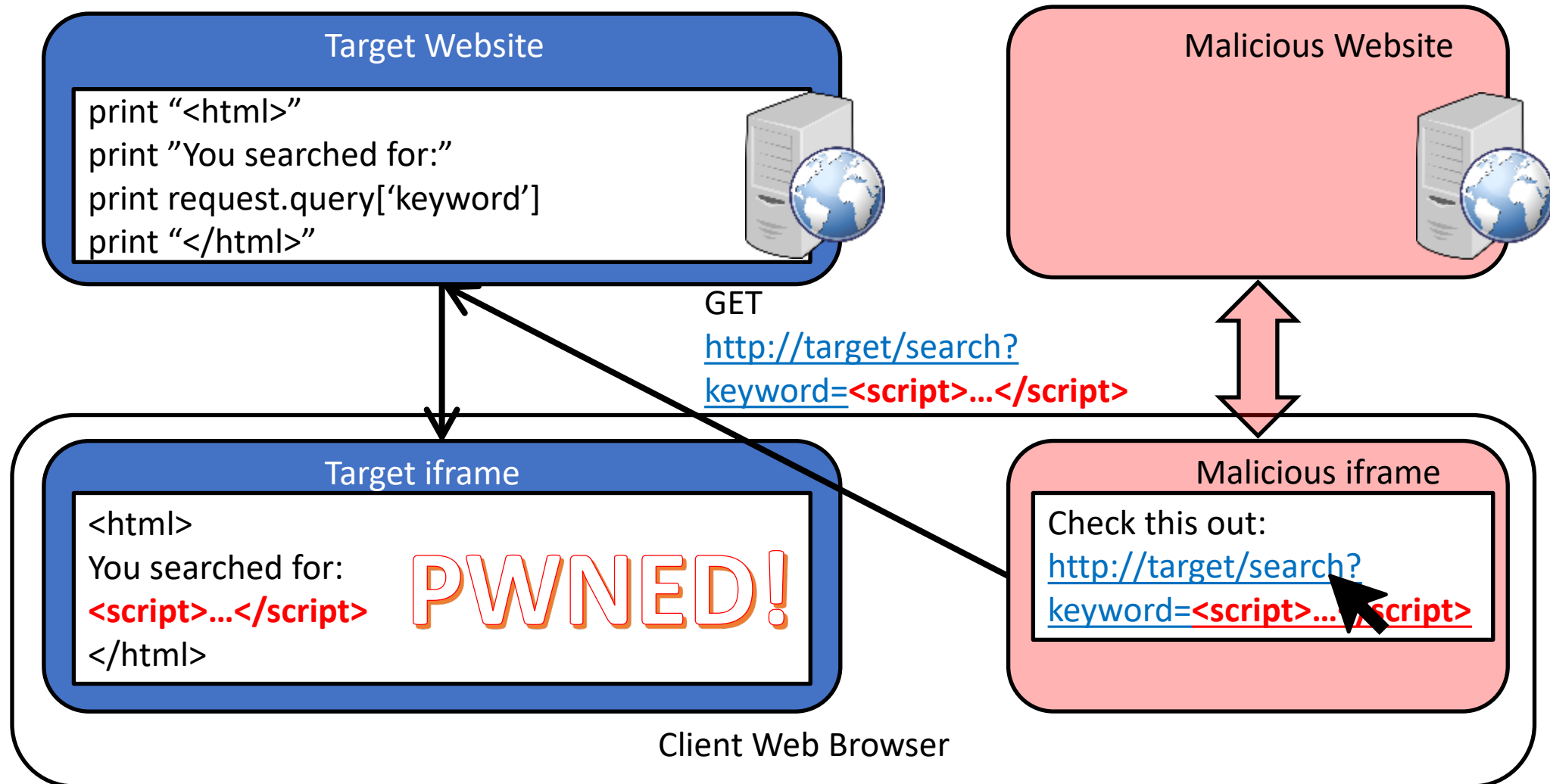
Browser Sandboxing – Same Origin Policy



Cross-Site Scripting

- Allows malicious website to execute (Java)script code
 - Across site boundaries
 - Ignoring SOP protections

Cross-Site Scripting



Cross-Site Scripting: Solution

- You need to properly sanitize user input:
[https://cheatsheetseries.owasp.org/cheatsheets/Cross Site Scripting Prevention Cheat Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html)
- Some sanitization libraries that can help you:
 - DOMPurify (for Node.js): <https://github.com/cure53/DOMPurify>
 - Python Bleach: <https://pypi.org/project/bleach/>
 - PHP HTML Purifier: <http://htmlpurifier.org/>

Insecure Object Reference Exploit

- Someone can access something by knowing where it is, despite not having proper security credentials
 - <http://bank.com/?account=9844>
 - <http://bank.com/?account=9845>
- Solution: check for credentials on every access!
 - E.g. Check session ID cookie to verify access is part of authenticated session

Buffer Overflow

- 2019 CWE (Common Weakness Enumeration) Top 25: Winner
 - Consistently within the top 3 for all years since 2009
- Reading or writing past the end of memory allocated for a buffer
 - Doesn't happen in Java (results in a `IndexOutOfBoundsException` exception)
 - Doesn't happen in JavaScript or Python (results in silent expansion of buffer)
 - Only happens in C / C++ / Assembly – allows direct access to memory
 - But a lot of critical system code is written in C / C++, unfortunately
- What Heartbleed was – see `heartbleed.c` in `sample_code` directory

heartbleed.c

```
void bad(int len) {
    char* notSecret = "open data";
    char* secret = "SECRET DATA HERE! NOBODY SHOULD SEE THIS!";
    printf("Sending data:\n");
    for (int j=0; j < len; j++) {
        printf("%c", notSecret[j]);
    }
}


int main() {
    int l;
    puts("Enter length of data:");
    scanf("%d", &l);
    bad(l);
}
```

```
-bash-4.1$ gcc heartbleed.c -o heartbleed
-bash-4.1$ ./heartbleed
Enter length of data:
100
Sending data:
open dataSECRET DATA HERE! NOBODY SHOULD SEE THIS!
Sending data:Enter length of data:%d
```


heartbleed.c --- Why?

- Assembly code generated from heartbleed.c:

```
.LC0:
    .string "open data"    notSecret[0]
.LC1:
    .string "SECRET DATA HERE! NOBODY SHOULD SEE THIS!"
.LC2:
    .string "Sending data:"
.LC3:
    .string "Enter length of data:"
.LC4:
    .string "%d"
bad(int):
    ...
```



Broken Authentication Exploit

- One user pretends to be another
- How?
 - Brute force guess or crack passwords
 - Do “Password reset” using personal info
 - Intercept unencrypted session IDs
- Apple iCloud had a leak suspected of being this
 - iCloud API allowed unlimited attempts – allowing a brute force attack

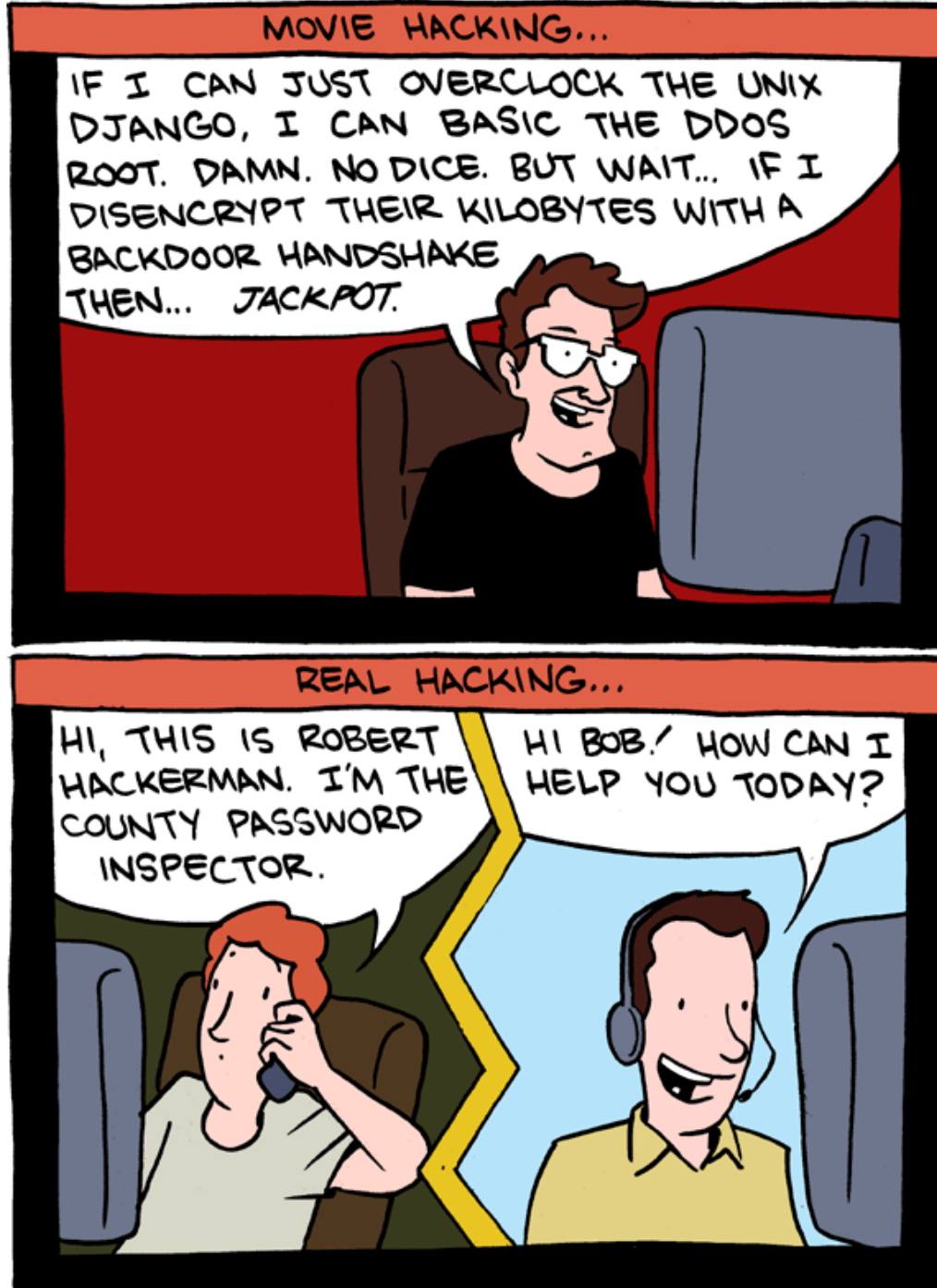
Security Misconfiguration Exploit

- Proper security is available, it's just not set up correctly!
- Examples
 - Default passwords
 - Firewalls with dangerous exceptions
 - File system directory listing in web server not disabled
 - Web server error log display left on (showing Java exception stack trace)

Insecure Storage Exploit

- Private data is stored in an unsecure way
- Examples
 - Credit card numbers stored in /tmp as part of logging all transactions
 - DB file with incorrect permissions, allowing DB file to be copied wholesale
 - Passwords in database not encrypted, or encrypted without salting

Social Engineering



For a More Comprehensive List ...

- CWE (Common Weakness Enumeration) Top 25:
 - https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html
 - By MITRE Corp. which maintains CVE (Common Vulnerabilities and Exposures) DB
- OWASP (Open Web Applications Security Project) Top 10 Project:
 - <https://owasp.org/www-project-top-ten/>
 - Top 10 security vulnerabilities for web applications over the years
- OWASP attacks page:
 - <https://owasp.org/www-community/attacks/>
 - Contains guides on how to test and mitigate for those vulnerabilities

Some Protections against Exploits

- Firewalls: a barrier between trusted intranet and untrusted internet
 - Can prevent access to network services such as FTP by untrusted party
 - Can prevent many types of DoS attacks on services by external party
- CDNs (Content Delivery Networks) – e.g. Akamai
 - Delivers content using many distributed servers around the world
 - Prevents even DDoS (Distributed DoS) attacks
- Cryptography – e.g. HTTPS protocol
 - Prevents unauthorized read or modification of data (e.g. HTTP packets)
- Operating system file permissions / Database table permissions
- Well-written code
- User training

Some Penetration Testing Tools

- Nmap – Network Mapper
 - Audits the network for open ports and open services
 - Audits version numbers for all OSes and services
- Wireshark – A packet sniffer that displays all network traffic
- Metasploit – A penetration testing tool for system vulnerabilities
 - Over 900 exploits for various OSes
 - Includes fuzzing technology to look for unknown software vulnerabilities
- Incidentally, these are the same tools attackers use
 - You might as well use them yourselves to test your systems

Pittsburgh – A Great City To Learn About Security!

- Many security researchers here at Pitt and CMU
 - LERSAIS at Pitt SCI: <http://www.sis.pitt.edu/lersais/>
 - Pitt Cyber Institute: <https://www.cyber.pitt.edu/home>
 - CyLab at CMU: <https://www.cylab.cmu.edu/>
- SEI (Software Engineering Institute): <https://www.sei.cmu.edu/>
- CERT (Computer Emergency Response Team):
<https://www.sei.cmu.edu/about/divisions/cert/>

Now Please Read Textbook Chapter 20

... and this ends all official lectures.

- Are you sad? Here are some bonus slides for you
- Note: Slides following will ***not*** appear in the exam

Bonus Security Slides

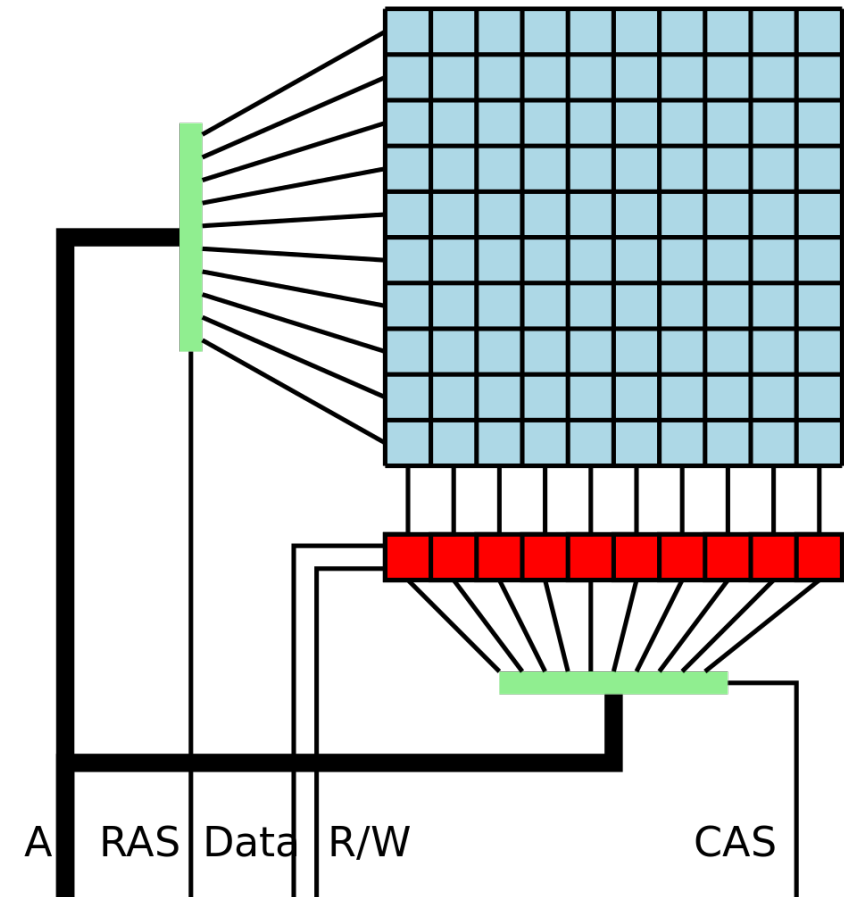
How far are exploits willing to go?

Row Hammer Exploit

Discovered by Google Project Zero, 2015

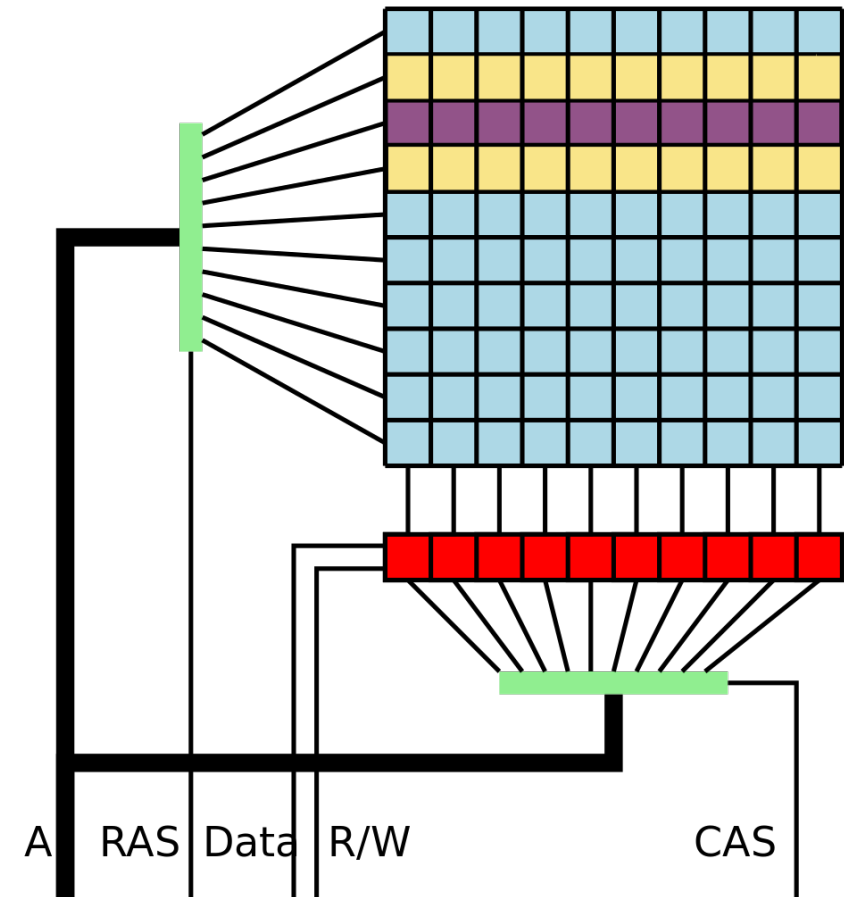
DRAM Organization

- Each square in matrix is a memory cell
 - Stores one bit of memory
 - Basically a capacitor holding a charge
- RAS (Row Address Strobe) selects row
 - Row is stored in row buffer (in red)
- CAS selects column, the specific bit
- Memory cell leaks charge over time
 - Needs refresh every 200 ms or so
 - Refresh recharges capacitors



Row Hammer Exploit

- Suppose **purple line** contains a password, the target of the exploit
- Keep hammering the neighboring **yellow lines** with reads
- Rapid voltage fluctuations of RAS lines cause **purple line** to lose charge faster
- Cells in **purple line** become zeroed out before getting a chance for refresh
- Password is now 00000000



Spectre & Its Root Causes

Paul Kocher
(paul@paulkocher.com)

ISCA

June 4, 2018



*If the surgery proves unnecessary, we'll
revert your architectural state at no charge.*

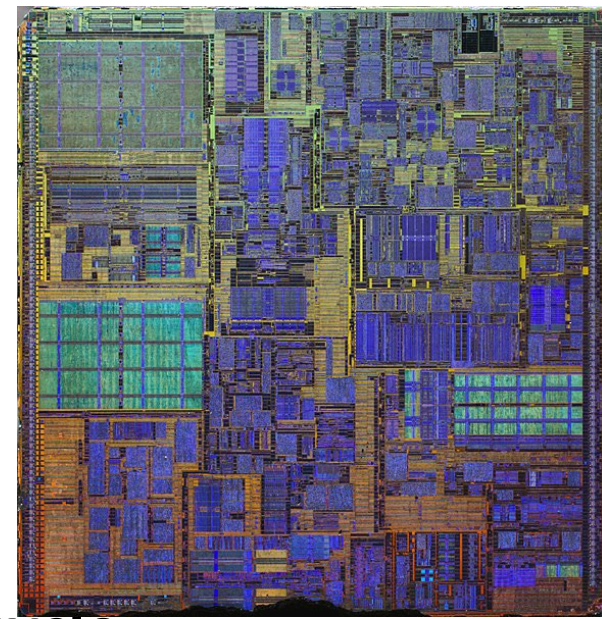
Addicted to speed

Performance goal

- Lowest time to reach the result same as running program in-order

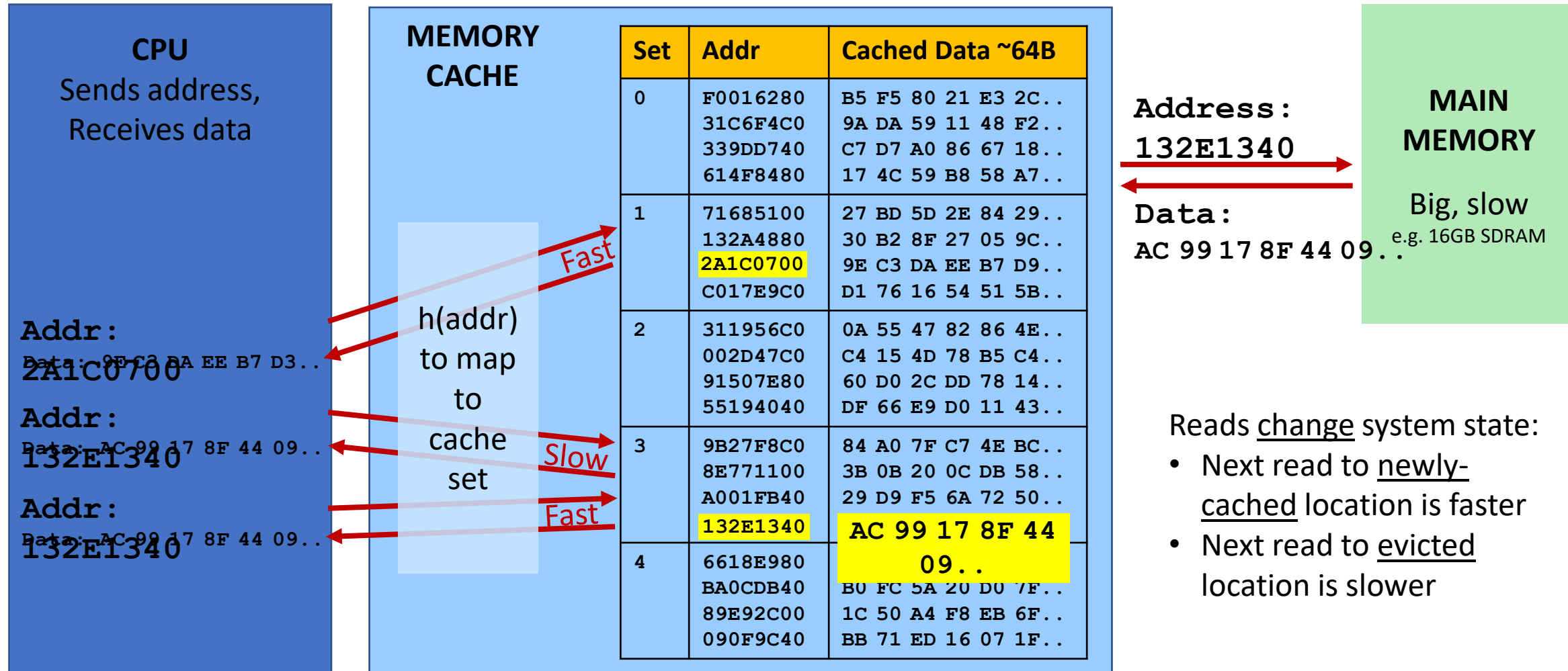
Single-thread speed gains require getting more done per clock cycle

- Memory latency is slow and not improving much
- Clock rates are maxed out: Pentium 4 reached 3.8 GHz in 2004
- How to do more per clock?
 - Reducing memory delays → Caches
 - Working during delays → Speculative execution



Memory caches for dummies

- Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



Speculative execution

Example of speculative execution:

```
if (uncached_value_usually_1 == 1)
    foo()
```

- ▶ Branch predictor: if() will probably be 'true' (based on prior history)
- ▶ CPU starts foo() speculatively -- but doesn't commit changes
- ▶ When value returns, changes committed only when value is actually '1'

Violates software security requirement that the CPUs runs instructions correctly.

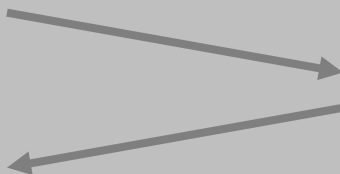
Regular execution

Set up the conditions so the processor will make a desired mistake

Fetch the sensitive data from the covert channel

Erroneous speculative execution

Mistake leaks sensitive data into a covert channel (e.g. state of the cache)



Conditional branch (Variant 1) attack

```
int flag = 0;
int array[MAX_SIZE * 4096], x, y;
if (flag)
    y = array[kernel_array[x]*4096];
```

- Attacker wants to get the value of `kernel_array[x]`. How?
- Attacker concocts and executes the above code.
- Note: the above code is completely legal
 - CPU will not read `array[kernel_array1[x]*4096]` unless `flag != 0`
- Uses speculative execution to furtively detect value of `kernel_array[x]`

Conditional branch (Variant 1) attack

```
if (flag)
    y = array[kernel_array[x]*4096];
```

Before attack:

- Train branch predictor to expect if() is true
- Evict `flag` and `array[]` from cache

Memory & Cache Status

`flag = 00000000`

Memory at `kernel_array` base address:
[... up to `kernel_array` base+x...]

09 F1 98 CC 90 ... (something secret)

array[0*4096]
array[1*4096]
array[2*4096]
array[3*4096]
array[4*4096]
array[5*4096]
array[6*4096]
array[7*4096]
array[8*4096]
array[9*4096]
array[10*4096]
array[11*4096]
...

Contents don't matter
only care about cache

status **Uncached**

Cached

Conditional branch (Variant 1) attack

```
if (flag)
    y = array[kernel_array[x]*4096];
```

Attacker does covert channel attack

- Speculative exec while waiting for `flag`
 - Predict that `if()` is true
 - Read address (`kernel_array` base + `x`)
 - Read returns secret byte = **09**
 - Request memory at (`array` base + **09***4096)
 - Brings `array[09*4096]` into the cache
 - Realize `if()` is false: discard speculative work
- Finish operation & return to caller

Attacker measures read time for `array[i*4096]`

- Read for `i=09` is fast (cached), revealing secret byte
- Repeat with many `x` (eg ~10KB/s)

Memory & Cache Status

`flag = 00000000`

Memory at `kernel_array` base address:
[... up to `kernel_array` base+`x`...]

09 F1 98 CC 90 ... (something secret)

`array[0*4096]`
`array[1*4096]`
`array[2*4096]`
`array[3*4096]`
`array[4*4096]`
`array[5*4096]`
`array[6*4096]`
`array[7*4096]`
`array[8*4096]`
`array[9*4096]`
`array[10*4096]`
`array[11*4096]`
...

Contents don't matter
only care about cache

status **Uncached**

Cached

Indirect branches (Variant 2)

Can go anywhere instantly (“jmp [rax]”)

- Poison predictor so victim speculative executes a ‘gadget’ that leaks memory

- Attack steps

- **Poison** branch predictor/BTB so speculative execution will go to gadget
- **Evict** from the cache or do other setup to encourage speculative execution
- **Execute** victim so it runs gadget speculatively
- **Read** sensitive data from covert channel
- **Repeat**

