Security

Nick Sumner
Security in General

- *Security*
  - Maintaining desired properties in the presence of adversaries
Security in General

- **Security**
  - Maintaining *desired properties* in the presence of adversaries

So what are the desired properties?
Security in General

- **Security**
  - Maintaining desired properties in the presence of adversaries

- CIA Model – classic security properties
Security in General

• **Security**
  – Maintaining desired properties in the presence of adversaries

• CIA Model – classic security properties
  – **Confidentiality**
    • Information is only disclosed to those authorized to know it
Security in General

- **Security**
  - Maintaining desired properties in the presence of adversaries

- CIA Model – classic security properties
  - Confidentiality
  - **Integrity**
    - Only modify information in **allowed ways by authorized parties**
Security in General

- **Security**
  - Maintaining desired properties in the presence of adversaries

- **CIA Model – classic security properties**
  - Confidentiality
  - **Integrity**
    - Only modify information in allowed ways by authorized parties
  - Do what is expected
Security in General

- **Security**
  - Maintaining desired properties in the presence of adversaries

- CIA Model – classic security properties
  - Confidentiality
  - Integrity
  - **Availability**
    - Those authorized for access are not prevented from it
Security in Software

- **Bugs** in software can lead to policy violations
  - Information leaks (C)
Security in Software

- *Bugs* in software can lead to policy violations
  - Information leaks (C)
  - Data Corruption (I)
Security in Software

- **Bugs** in software can lead to policy violations
  - Information leaks (C)
  - Data Corruption (I)
  - Denial of service (A)
Security in Software

• **Bugs** in software can lead to policy violations
  – Information leaks (C)
  – Data Corruption (I)
  – Denial of service (A)
  – Remote execution – (CIA) arbitrarily bad!
Security in Software

- *Bugs* in software can lead to policy violations
- *Bugs* make software vulnerable to attack
Security in Software

- **Bugs** in software can lead to policy violations
- **Bugs** make software vulnerable to attack
  - XSS
  - SQL Injection
  - Buffer overflow
  - Path replacement
  - Integer overflow
  - Race conditions (TOCTOU – Time of Check to Time of Use)
  - Unsanitized format strings
  - ...

All create attack vectors for a malicious adversary
Why Is This Special?

Poor security comes from unintended behavior.
→ Quality software shouldn't allow such actions anyway.
Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

• While our testing techniques so far find some security issues, many slip through! Why?
Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

- **While our testing techniques so far find some security issues, many slip through! Why?**
  - We cannot test everything
Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

- While our testing techniques so far find some security issues, many slip through! Why?
  - We cannot test everything
  - Concessions form part of an attack surface
    - Networks, Software, People
Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

- While our testing techniques so far find some security issues, many slip through! Why?
  - We cannot test everything
  - Concessions form part of an *attack surface*
    - Networks, Software, People

- **Need additional policies & testing methods that specifically address security**
What Could Possible Go Wrong?

- Many ways to attack different programs
- MITRE groups the most common into:
What Could Possible Go Wrong?

- Many ways to attack different programs
- MITRE groups the most common into:
  - Insecure Interaction
    - Data sent between components in an insecure fashion
What Could Possible Go Wrong?

- Many ways to attack different programs
- MITRE groups the most common into:
  - Insecure Interaction
    - Data sent between components in an insecure fashion
  - Risky Resource Management
    - Bad creation, use, transfer, & destruction of resources
What Could Possible Go Wrong?

- Many ways to attack different programs
- MITRE groups the most common into:
  - Insecure Interaction
    - Data sent between components in an insecure fashion
  - Risky Resource Management
    - Bad creation, use, transfer, & destruction of resources
  - Porous Defenses
    - Standard security practices that are missing or incorrect

[http://cwe.mitre.org/top25/#Categories]
Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - [Memory Safety](http://www.pl-enthusiast.net/2014/07/21/memory-safety/)
Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- Provide common attack patterns [Eternal War in Memory]
Memory Safety

- **Unsafe memory** accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]

- **Provide common attack patterns** [Eternal War in Memory]

```
Dangling or OOB * → Read or Write
```
Memory Safety

- **Unsafe memory** accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]

- **Provide common attack patterns** [Eternal War in Memory]
Memory Safety

- **Unsafe memory** accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- **Provide common attack patterns** [Eternal War in Memory]

![Diagram showing relationships between data, code, and memory corruption]

- Data *
- Code
- Code Corruption
- Dangling or OOB *
- Read or Write

[Diagram relationships between data, code, and memory corruption]
Memory Safety

- **Unsafe memory** accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]

- **Provide common attack patterns** [Eternal War in Memory]

![Diagram showing memory safety concepts](image-url)
Memory Safety

- **Unsafe memory** accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- **Provide common attack patterns** [Eternal War in Memory]
Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- *Provide common attack patterns* [Eternal War in Memory]
Code Corruption

def foo():
    # original code
    ...

def foo():
    # malicious code
    ...

• How can we prevent this?
Code Corruption

- How can we prevent this?
- What problems does this solution create?
Control Flow Hijacking

```c
void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}
```

How many of you recall what a stack frame looks like?
Data Only Attacks

```c
void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}
```
Data Only Attacks

void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}
Data Only Attacks

void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}

Addresses

Stack Growth

Stack frame for foo
Data Only Attacks

void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}

Addresses

Stack Growth

0xFFF

Stack

... Previous Frame

Return Address

Old Frame Ptr

secureData

buffer[15]

buffer[14]

...

buffer[0]
Data Only Attacks

What can go wrong?

```c
void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}
```
Data Only Attacks

void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}

input = “normal input” + “insecureData”
Data Only Attacks

void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}

input = "normal input" + "insecureData"

buffer overflow attack
Data Only Attacks

void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}

input = "normal input" + "insecureData"

The integrity of the secure data is corrupted.
Control Flow Hijacking

void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}

Addresses

Stack Growth

Stack

Previous Frame

Return Address

Old Frame Ptr

secureData

buffer[15]

buffer[14]

...

buffer[0]
Control Flow Hijacking

void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}

input = "input"
    + "payload address"
    + "payload (shell code)"
Control Flow Hijacking

void foo(char *input) {
    unsigned secureData;
    char buffer[16];
    strcpy(buffer, input);
}

input = “input”
+ “payload address”
+ “payload (shell code)”

On return, we'll execute the shell code
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries

Abort because canary changed!
Control Flow Hijacking

• How can we prevent this basic approach?
  – Stack Canaries
  – DEP – Data Execution Prevention / $W \oplus X$
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
  - DEP – Data Execution Prevention / W⊕X

shell code:
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
  - DEP – Data Execution Prevention / W ⊕ X

Shell code:

<table>
<thead>
<tr>
<th>Previous Frame</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Canary</td>
<td></td>
</tr>
<tr>
<td>Old Frame Ptr</td>
<td></td>
</tr>
<tr>
<td>secureData</td>
<td></td>
</tr>
<tr>
<td>buffer[15]</td>
<td></td>
</tr>
<tr>
<td>buffer[14]</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>buffer[0]</td>
<td></td>
</tr>
</tbody>
</table>

Abort because W but not X
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
  - DEP – Data Execution Prevention / W⊕X

But these are still easily bypassed!
Return to libc Attacks

- Reuse existing code to bypass $W \oplus X$
Return to libc Attacks

- Reuse existing code to bypass $W \oplus X$

```
<table>
<thead>
<tr>
<th>Previous Frame</th>
<th>Fake Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
<td></td>
</tr>
<tr>
<td>Old Frame Ptr</td>
<td>Old Frame Ptr</td>
</tr>
<tr>
<td>secureData</td>
<td>secureData</td>
</tr>
<tr>
<td>buffer[14]</td>
<td>buffer[14]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>buffer[0]</td>
<td>buffer[0]</td>
</tr>
</tbody>
</table>
```

```
“/usr/bin/minesweeper”
```
Return to libc Attacks

- Reuse existing code to bypass $W \oplus X$

<table>
<thead>
<tr>
<th>Previous Frame</th>
<th>Fake Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
<td>&quot;system()&quot;</td>
</tr>
<tr>
<td>Old Frame Ptr</td>
<td></td>
</tr>
<tr>
<td>secureData</td>
<td></td>
</tr>
<tr>
<td>buffer[15]</td>
<td></td>
</tr>
<tr>
<td>buffer[14]</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>buffer[0]</td>
<td></td>
</tr>
</tbody>
</table>

```
SecureData
Buffer[15]
Buffer[14]
...
Buffer[0]
```

```
SecureData
Buffer[15]
Buffer[14]
...
Buffer[0]
```

"/usr/bin/minesweeper"

Even construct new functions piece by piece!
Return to libc Attacks

• Reuse existing code to bypass $W \oplus X$

• Return Oriented Programming
  – Build new functionality from pieces of existing functions
Return to libc Attacks

- Reuse existing code to bypass W⊕X
- Return Oriented Programming
  - Build new functionality from pieces of existing functions
**ASLR**

- Address Space Layout Randomization
  - You can't use it if you can't find it!

![Diagram showing ASLR with stack, heap, library, and program layers for two runs](image)
Address Space Layout Randomization

- You can't use it if you can't find it!

But even this is “easily” broken
Control Flow Integrity

- Restrict indirect control flow to needed targets
  - \texttt{Jmp */call */ret}

\begin{verbatim}
foo = ... 
foo();
\end{verbatim}
Control Flow Integrity

- Restrict indirect control flow to needed targets
  - Jmp */call */ret

```c
foo = ...
if foo not in [...] abort()
foo();
```

```c
void a() {
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
}
```

```c
void b() {
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
}
```
Control Flow Integrity

- Restrict indirect control flow to needed targets
  - Jmp */call */ret

```c
foo = ... 
if foo not in [...] abort() 
foo();
```

```c
void a() {
  ...  
} 
void b() {
  ...  
}
```

```bash
clang -flto -fsanitize=cfi -fsanitize=safe-stack
clang -fsanitize=safe-stack
```
Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers
Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers

- Why doesn't Java face this issue?
Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers
- Why doesn't Java face this issue?
- Is this intrinsic to languages like C++?
  - Why/Why not?
Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers

- Why doesn't Java face this issue?

- Is this intrinsic to languages like C++?
  - Why/Why not?

- Are these still a real issue?
Memory Safety

• Vulnerabilities come from reading/writing/freeing
  – Out of bounds pointers
  – Dangling pointers

• Why doesn't Java face this issue?

• Is this intrinsic to languages like C++?
  – Why/Why not?

• Are these still a real issue?
  – http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0015
  – http://seclists.org/oss-sec/2016/q1/645
  – ...
Root Causes Over Time

Root cause of CVEs by patch year


Stack Corruption  Heap Corruption  Use After Free  Type Confusion  Uninitialized Use  Heap OOB Read  Other

[Matt Miller – BlueHat 2019]
Another Case: SQL Injection

SQL – a query language for databases

- Queries like:
  "SELECT grade, id FROM students WHERE name="" + username;
Another Case: SQL Injection

SQL – a query language for databases

- Queries like:
  “SELECT grade, id FROM students
  WHERE name="" + username;

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alice</td>
<td>92</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>87</td>
</tr>
<tr>
<td>2</td>
<td>Mallory</td>
<td>75</td>
</tr>
</tbody>
</table>
Another Case: SQL Injection

SQL – a query language for databases

- Queries like:
  “SELECT grade, id FROM students
  WHERE name = ” + username;

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alice</td>
<td>92</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>87</td>
</tr>
<tr>
<td>2</td>
<td>Mallory</td>
<td>75</td>
</tr>
</tbody>
</table>

- Values for name, grade often come from user input.
Another Case: SQL Injection

SQL – a query language for databases

• Queries like:
  “SELECT grade, id FROM students
  WHERE name = ” + username;

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alice</td>
<td>92</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>87</td>
</tr>
<tr>
<td>2</td>
<td>Mallory</td>
<td>75</td>
</tr>
</tbody>
</table>

• Values for name, grade often come from user input.

Why is this a problem?
Another Case: SQL Injection

username = "'bob'; DROP TABLE students"

• What happens?
The user may include commands in their input!
SQL Injection

- The user may include commands in their input!
- Need to *sanitize* the input before use

SQL Injection


- The user may include commands in their input!
- Need to sanitize the input before use

How would you prevent this problem?
SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs
SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs

```java
List<Person>; people = //user input
Connection connection = DriverManager.getConnection(...);
connection.setAutoCommit(false);
try {
    PreparedStatement statement = connection.prepareStatement(
        "UPDATE people SET lastName = ?, age = ?, WHERE id = ?",
    );
    for (Person person : people){
        statement.setString(1, person.getLastName());
        statement.setInt(2, person.getAge());
        statement.setInt(3, person.getId());
        statement.execute();
    }
    connection.commit();
} catch (SQLException e) {
    connection.rollback();
}
```
Do not write raw SQL. (examples from bobby-tables.com)

- Sanitizing APIs

```java
EntityManager em = getEntityManager();
Query query = em.createNativeQuery("SELECT E.* from EMP E, ADDRESS A WHERE E.EMP_ID = A.EMP_ID AND A.CITY = ?", Employee.class);
query.setParameter(1, "Ottawa");
List<Employee> employees = query.getResultList();
```
SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs
  - ORMs (to some degree!) [Fixing SQL Injection w/ Hibernate]
SQL Injection

• Do not write raw SQL. (examples from bobby-tables.com)
  – Sanitizing APIs
  – ORMs (to some degree!) [Fixing SQL Injection w/ Hibernate]

String name = //user input
int age = //user input
Session session = //...
Query query = session.createQuery("from People where lastName = :name and age > :age");
query.setString("name", name);
query.setInteger("age", age);
Iterator people = query.iterate();
SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs
  - ORMs (to some degree!) [Fixing SQL Injection w/ Hibernate]

```java
String name = //user input
int age = //user input
Session session = //...
Query query = session.createQuery(
    "from People where lastName = :name and age > :age";
query.setString("name", name);
query.setInteger("age", age);
Iterator people = query.iterate();
```

- Use abstractions that design error away if possible!
  - Applies whenever you generate code in another language (think web apps)
Side Channels

• So far we have looked for ways to *directly* violate CIA
Side Channels

- So far we have looked for ways to *directly* violate CIA
  - Execute code
  - Explicitly broadcast a value
  - ...
Side Channels

- So far we have looked for ways to *directly* violate CIA
  - Execute code
  - Explicitly broadcast a value
  - ...

- An attacker *indirectly* violate CIA by inferring sensitive information
Side Channels

• So far we have looked for ways to directly violate CIA
  – Execute code
  – Explicitly broadcast a value
  – ...

• An attacker can indirectly violate CIA by inferring sensitive information
  – *Side channel attacks* can infer secret information about a system based on implementation details
Side Channels

- So far we have looked for ways to directly violate CIA
  - Execute code
  - Explicitly broadcast a value
  - ...

- An attacker can indirectly violate CIA by inferring sensitive information
  - *Side channel attacks* can infer secret information about a system based on implementation details
  - These leaks can be present even for algorithms that are mathematically correct
Side Channels

- So far we have looked for ways to *directly* violate CIA
  - Execute code
  - Explicitly broadcast a value
  - ...

- An attacker can *indirectly* violate CIA by inferring sensitive information
  - *Side channel attacks* can infer secret information about a system based on implementation details
  - These leaks can be present even for algorithms that are mathematically correct
  - Leaks can come from: Output, Timing (compute, cache, MDS,...), Power, Sound, Light, ...
Side Channels

- Consider code that directly leaks a sensitive boolean

```python
def very_stupid(greeting, sensitive):
    ...
    log_to_nonsensitive(sensitive)
    ...
```
Side Channels

- Consider code that directly leaks a sensitive boolean

```python
def very_stupid(greeting, sensitive):
    ...
    log_to_nonsensitive(sensitive)
    ...
```

- This could be tweaked to become an indirect leak

```python
def still_bad(greeting, sensitive):
    ...
    if sensitive:
        log_to_nonsensitive(greeting)
    ...
```
Side Channels

- Consider code that directly leaks a sensitive boolean

```python
def very_stupid(greeting, sensitive):
    ...
    log_to_nonsensitive(sensitive)
    ...
```

- This could be tweaked to become an indirect leak

```python
def still_bad(greeting, sensitive):
    ...
    if sensitive:
        log_to_nonsensitive(greeting)
    ...
```

- The **value** of the sensitive information can be inferred by the **existence** of the nonsensitive information!
Side Channels

- Any difference in behavior between sensitive and nonsensitive tasks can be measured and used
Any difference in behavior between sensitive and nonsensitive tasks can be measured and used

```python
def subtly_bad(greeting, sensitive):
    ...
    if sensitive:
        expensive_computation()
        log_to_nonsensitive(greeting)
    ...
```
Side Channels

- Any difference in behavior between sensitive and nonsensitive tasks can be measured and used

```python
def subtly_bad(greeting, sensitive):
    ...
    if sensitive:
        expensive_computation()
    log_to_nonsensitive(greeting)
    ...
```

This has been the downfall of crypto implementations!
Any difference in behavior between sensitive and nonsensitive tasks can be measured and used

```python
def subtly_bad(greeting, sensitive):
    ...
    if sensitive:
        expensive_computation()
        log_to_nonsensitive(greeting)
    ...
```

```python
def deviously_bad(greeting, sensitive):
    ...
    if sensitive:
        a[not_in_cache] = ...
        log_to_nonsensitive(greeting)
    ...
```
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```python
if x < array1.size:
    y = array2[array1[x] * 4096]
```
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```python
if x < array1.size:
    y = array2[array1[x] * 4096]
```
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```python
if x < array1.size:
y = array2[array1[x] * 4096]
```

When the condition is `true`, `array1[x]` will be in bounds
Side Channels

• This is the fundamental premise behind Spectre and generic MDS based attacks
  – Spectre worked by mistraining speculation & then measuring timing differences

When the condition is true, array1[x] will be in bounds
When the condition is false, array1[x] can be anywhere
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```python
if x < array1.size:
    y = array2[array1[x] * 4096]
```

When the condition is `true`, `array1[x]` will be in bounds
When the condition is `false`, `array1[x]` can be anywhere

An attacker can
Side Channels

• This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```python
if x < array1.size:
    y = array2[array1[x] * 4096]
```

When the condition is true, `array1[x]` will be in bounds

When the condition is false, `array1[x]` can be anywhere

An attacker can
1) make `array1[x]` point to sensitive data
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```plaintext
if x < array1.size:
y = array2[array1[x] * 4096]
```

When the condition is true, `array1[x]` will be in bounds
When the condition is false, `array1[x]` can be anywhere

An attacker can
1) make `array1[x]` point to sensitive data
2) train the branch to speculate true
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences
    
    if \( x < \) array1.size:
    
    \[
    y = \text{array2[array1[x]]} * 4096
    \]

    When the condition is \textit{true}, array1[x] will be in bounds
    When the condition is \textit{false}, array1[x] can be anywhere

    The sensitive data is speculatively read and used!

    An attacker can
    1) make \text{array1[x]} point to sensitive data
    2) train the branch to speculate \textit{true}
• This is the fundamental premise behind Spectre and generic MDS based attacks
  – Spectre worked by mistraining speculation & then measuring timing differences

\[
\text{if } x < \text{array1.size:} \\
  y = \text{array2[array1[x] * 4096]}
\]

When the condition is \textit{true}, \text{array1[x]} will be in bounds

When the condition is \textit{false}, \text{array1[x]} can be anywhere

An attacker can
1) make \text{array1[x]} point to sensitive data
2) train the branch to speculate \textit{true}
3) extract the data through a 1-hot encoding in the time to access elements of \text{array2}
   (or a buffer sharing the cache mapping of \text{array2})
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```python
if x < array1.size:
y = array2[array1[x] * 4096]
```

```python
# foo is a function pointer
foo()
```

Foo can be trained to speculate to an arbitrary gadget!
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```python
if x < array1.size:
    y = array2[array1[x] * 4096]

# foo is a function pointer
foo()

def foo():
    return
```

Return targets can be trained to speculate to gadgets!
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```python
if x < array1.size:
    y = array2[array1[x] * 4096]

# foo is a function pointer
foo()

def foo():
    return
```

Note: This means that ROP gadgets can once again be used! Newer compiler options can mitigate but not remove the challenge
Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences
    
    ```python
    if x < array1.size:
        y = array2[array1[x] * 4096]
    ```

    # foo is a function pointer
    foo()

    ```python
    def foo():
        return
    ```

  - MDS attacks leverage other CPU artifacts to achieve similar goals (line buffers, ports, etc.)
    - Contention on any resource affects timing
A Subtle Problem in General

- The problems may be much more subtle:

  User A can read files X,Y,Z and write to S,T
  User B can read files X,Y,S and write to Z,T
A Subtle Problem in General

The problems may be much more subtle:

User A can read files X,Y,Z and write to S,T
User B can read files X,Y,S and write to Z,T

How can we ensure that no information from A is ever written to Z?
A Subtle Problem in General

- The problems may be much more subtle:

  User A can read files X,Y,Z and write to S,T
  User B can read files X,Y,S and write to Z,T

  How can we ensure that no information from A is ever written to Z?

  Can you envision a scenario that creates this problem?
A Subtle Problem in General

• The problems may be much more subtle:

  User A can read files X,Y,Z and write to S,T
  User B can read files X,Y,S and write to Z,T

  How can we ensure that no information from A is ever written to Z?

• Care may be required to enforce access control policies
A Subtle Problem in General

- The problems may be much more subtle:
  
  User A can read files X,Y,Z and write to S,T  
  User B can read files X,Y,S and write to Z,T

  How can we ensure that no information from A is ever written to Z?

- Care may be required to enforce *access control policies*
  
  - *Discretionary access control* – owner determines access
A Subtle Problem in General

- The problems may be much more subtle:

  User A can read files X,Y,Z and write to S,T
  User B can read files X,Y,S and write to Z,T

  How can we ensure that no information from A is ever written to Z?

- Care may be required to enforce access control policies
  - Discretionary access control – owner determines access
  - Mandatory access control – clearance determines access
Assuring Security

• Make risky operations someone else's job
  – e.g. Google Checkout, PayPal, Amazon, etc.
Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
- Follow secure design & coding policies
  - And include them in your review criteria
Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
- **Follow secure design & coding policies**
  - And include them in your review criteria
  - Apple secure coding policies
  - CERT Top 10 Practices
  - Mitre Mitigation Strategies
Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
- Follow secure design & coding policies
  - And include them in your review criteria
- Formal certification
Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
- Follow secure design & coding policies
  - And include them in your review criteria
- Formal certification
- Follow established security workflows (OWASP, BSIMM, ...)
Common Proactive Approaches

How are these techniques applied?
Common Proactive Approaches

How are these techniques applied?

- Security must be part of design
  - Prepared Statements, Safe Arrays, etc.
Common Proactive Approaches

How are these techniques applied?

- Security must be part of design
  - Prepared Statements, Safe Arrays, etc.

- Regular security audits
  - Retrospective analysis & suggestions
Common Proactive Approaches

How are these techniques applied?

• Security must be part of design
  – Prepared Statements, Safe Arrays, etc.

• Regular security audits
  – Retrospective analysis & suggestions

• Penetration testing (Pen Testing)
  – Can someone skilled break it?
When you find a vulnerability

- Reporting security vulnerabilities is good
When you find a vulnerability

- Reporting security vulnerabilities is good
- Making them public immediately is not
When you find a vulnerability

- Reporting security vulnerabilities is good
- Making them public immediately is not
- \textit{Responsible disclosure} policies govern the trade off between allowing a fix to be deployed & awareness
When you find a vulnerability

- Reporting security vulnerabilities is good
- Making them public immediately is not
- Responsible disclosure policies govern the trade off between allowing a fix to be deployed & awareness
  - e.g. Google standard 90 day window
  - 7 month window for Spectre due to severity
  ...
Security Overall

- Security is now a pressing concern for all software
Security Overall

- Security is now a pressing concern for all software
  - Old software was designed in an era of naiveté and is often vulnerable/broken
Security Overall

- Security is now a pressing concern for all software
  - Old software was designed in an era of naiveté and is often vulnerable/broken
  - New software is built to perform sensitive operations in a multiuser and networked environment.
Security Overall

- Security is now a pressing concern for all software
  - Old software was designed in an era of naïveté and is often vulnerable/broken
  - New software is built to perform sensitive operations in a multiuser and networked environment.

Not planning for security concerns from the beginning is a broken approach to development