CMPT 745
Software Engineering

Software Security

Nick Sumner
wsumner@sfu.ca
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

Establishing *authenticity* is a part.
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 Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance
Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance

- Requires an understanding of
  - how attacks may occur and policies for prevention
Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance

- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance

- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
  - how to recognize attacks
Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance

- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
  - how to recognize attacks
  - how to mitigate attacks in progress
Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance

- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
  - how to recognize attacks
  - how to mitigate attacks in progress
  - how to adapt to prevent future attacks
Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance

- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
  - how to recognize attacks
  - how to mitigate attacks in progress
  - how to adapt to prevent future attacks

- These can be interpreted to extend far beyond software systems (spearphishing, physical theft, ...)
  - We will focus on software & related security aspects
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security

“The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards - and even then I have my doubts.”

- Gene Spafford
Security in Software Development

• **Big picture: Security is not Boolean**
  – You cannot achieve perfect security
  – You must assess, prioritize, and manage security *risks* over time
  • What do you value?
Security in Software Development

• **Big picture: Security is not Boolean**
  – You cannot achieve perfect security
  – You must assess, prioritize, and manage security *risks* over time
    • What do you value?
    • What are your CIA liabilities?
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security risks over time
    - What do you value?
    - What are your CIA liabilities?
    - What threatens them?
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
    - What do you value?
    - What are your CIA liabilities?
    - What threatens them?
    - Who threatens them & with what power?
Security in Software Development

• **Big picture: Security is not Boolean**
  – You cannot achieve perfect security
  – You must assess, prioritize, and manage security *risks* over time
    • What do you value?
    • What are your CIA liabilities?
    • What threatens them?
    • Who threatens them & with what power?
    • How can you defend against them? Where can you break an attack chain?
Security in Software Development

• **Big picture: Security is not Boolean**
  – You cannot achieve perfect security
  – You must assess, prioritize, and manage security *risks* over time
  – Classically: Risk = E[Loss]
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security risks over time
  - Classically: Risk = $E[\text{Loss}] = \text{Impact} \times \text{Probability}$
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = $E[\text{Loss}] = \text{Impact} \times \text{Probability}$

\[
\text{Vulnerability} \times \text{Threat}
\]
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security risks over time
  - Classically: Risk = E[Loss] = Impact × Probability

\[ \text{Vulnerability} \times \text{Threat} \]

A weakness in a system that can cause harm
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = $E[\text{Loss}] = \text{Impact} \times \text{Probability}$

A weakness in a system that can cause harm

Action by an adversary, using a vulnerability to cause harm
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security risks over time
  - Classically: Risk = E[Loss] = Impact × Probability

- Vulnerability × Threat

A weakness in a system that can cause harm Action by an adversary, using a vulnerability to cause harm

Think back to our discussions on performance analysis. Why is this inadequate?
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: \( \text{Risk} = \mathbb{E}[\text{Loss}] = \text{Impact} \times \text{Probability} \)

\[
\text{Vulnerability} \times \text{Threat}
\]

A weakness in a system that can cause harm

Action by an adversary, using a vulnerability to cause harm

Think back to our discussions on performance analysis. Why is this inadequate?

These dangers in assessment apply to all good engineering
Security in Software Development

- **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = $E[\text{Loss}] = \text{Impact} \times \text{Probability}$
  - Good risk analysis requires clear identification of all actors in the formula
Security in Software Development

● **Big picture: Security is not Boolean**
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability
  - Good risk analysis requires clear identification of all actors in the formula
  - **Cost-Benefit analysis should guide decisions informed by risk**
Security in Software Development

- What will we cover?
Security in Software Development

- What will we cover?
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation
Security in Software Development

• **What will we cover?**
  – Common common threats & vulnerabilities
    • Data corruption
    • Information leaks (& side channels)
    • Privilege escalation
  – Approaches for finding potential vulnerabilities
    • Fuzz testing
Security in Software Development

- What will we cover?
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation
  - Approaches for finding potential vulnerabilities
    - Fuzz testing
  - Designing secure software
Security in Software Development

- **What will we cover?**
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation
  - Approaches for finding potential vulnerabilities
    - Fuzz testing
  - Designing secure software
  - **Defending against attackers**
    - Program transformation & hardening
Security in Software Development

- **What will we cover?**
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation
  - Approaches for finding potential vulnerabilities
    - Fuzz testing
  - Designing secure software
  - Defending against attackers
    - Program transformation & hardening
  - Reverse engineering & binary analysis
Thinking About Threats, Vulnerabilities, & Exploits
Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
Before exploring specific attacks, we must understand security goals & abstract ways attackers behave

Security goals come from the CIA triad
Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave

- **Security goals come from the CIA triad**
  - What information should be confidential?
  - Who are the authenticated parties?
  - What should they be able to access?
  - When?
Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad
- A threat model defines the potential threats & attack vectors to protect against
Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad
- A threat model defines the potential threats & attack vectors to protect against
  - Good threat modeling requires a “security mindset”
    Consider how things can be made to fail. [Schneier 2008]
Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave.

- Security goals come from the CIA triad.

- A *threat model* defines the potential threats & attack vectors to protect against.
  - Good threat modeling requires a “security mindset”.
    Consider how things can be made to fail. [Schneier 2008]

- Several approaches to threat modeling (Diagrams, trees, checklists, ...).
Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad
- A threat model defines the potential threats & attack vectors to protect against
  - Good threat modeling requires a “security mindset”
    Consider how things can be made to fail. [Schneier 2008]
- Several approaches to threat modeling (Diagrams, trees, checklists, ...)
  - STRIDE:
    Spoofing, Tampering, Repudiation, Info leaks, DOS, Escalated privileges
A Simple (Classic) Example

- Consider a paid compilation service
A Simple (Classic) Example

- Consider a paid compilation service
A Simple (Classic) Example

- Consider a paid compilation service

```
“(Foo.c,Bar.o)”
```
A Simple (Classic) Example

- Consider a paid compilation service
A Simple (Classic) Example

- Consider a paid compilation service
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- spoofing requests
- repudiate requests
- MITM tamper
- leak
- block
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a **confused deputy**
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a **confused deputy**
- privilege escalation is implicit
A Simple (Classic) Example

• Consider a paid compilation service
• What threats should we model? (CIA & STRIDE)

• The service must be allowed to update the bill.
• All requests execute with the authority of the service!

The service is a confused deputy
• privilege escalation is implicit

Can be addressed with capability based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a **confused deputy**
- privilege escalation is implicit

Can be addressed with **capability** based access control
A Simple (Classic) Example

• Consider a paid compilation service
• What threats should we model? (CIA & STRIDE)

The service must be allowed to update the bill.
All requests execute with the authority of the service!

The service is a confused deputy
• privilege escalation is implicit

Can be addressed with capability based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a *confused deputy*
- privilege escalation is implicit

Can be addressed with *capability* based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a confused deputy
- privilege escalation is implicit

Can be addressed with capability based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a confused deputy
- privilege escalation is implicit

Can be addressed with capability based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

The service is a confused deputy.
- privilege escalation is implicit

Can be addressed with capability based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

The service must be allowed to update the bill.
All requests execute with the authority of the service!
The service is a confused deputy
- privilege escalation is implicit

Can be addressed with capability based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a confused deputy
- privilege escalation is implicit

Can be addressed with capability based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a confused deputy
- privilege escalation is implicit

Can be addressed with capability based access control
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

The service must be allowed to update the bill.
All requests execute with the authority of the service!

The service is a confused deputy
privilege escalation is implicit

Can be addressed with capability based access control

Clients

Compiler Service

```
﹃(Foo.c,bill.txt)﹄
```

Blocked by OS!

```
Hello! My name is: Mallory
```

Outputs
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

NOTE:
We deal every day with a very confused deputy: web browsers
CSRF, Clickjacking, XSS, ...
Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
  - Information leaks
  - Data corruption
  - Denial of service
Low Level Vulnerabilities

- **Within software, bugs can lead to vulnerabilities**
  - Information leaks
  - Data corruption
  - Denial of service
  - Remote code execution! ... !!
Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
- Bugs make software vulnerable to attack
Low Level Vulnerabilities

• Within software, bugs can lead to vulnerabilities

• Bugs make software vulnerable to attack
  – Buffer overflow
  – Path replacement
  – Integer overflow
  – Race conditions (TOCTOU – Time of Check to Time of Use)
  – Unsanitized format strings
  – ...

Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
- Bugs make software vulnerable to attack
  - Buffer overflow
  - Path replacement
  - Integer overflow
  - Race conditions (TOCTOU – Time of Check to Time of Use)
  - Unsanitized format strings
  - ...

All create attack vectors for an adversary.
Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
- Bugs make software vulnerable to attack
  - Buffer overflow
  - Path replacement
  - Integer overflow
  - Race conditions (TOCTOU – Time of Check to Time of Use)
  - Unsanitized format strings
  - ...
- We will specifically look at issues of *memory safety* and *side channels*
Unsafe memory accesses are a longstanding vector
- Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
Software Security

- **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]
Software Security

- *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]
Software Security

- *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]
Software Security

- *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]
Software Security

- **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]
Software Security

- **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]
Software Security

- **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

• How can we prevent this?
Code Corruption

def foo():
    # original code
    ...

def foo():
    # malicious code
    ...

- How can we prevent this?
Code Corruption

- How can we prevent this?
Code Corruption

- How can we prevent this?
- What problems could this solution create?

(Might you want executable data?)
Control Flow Hijacking

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

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

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

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

Stack frame for `foo`

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

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

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

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

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

On return, we'll execute the shell code

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

input = “input”
+ “payload address”
+ “payload (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

<table>
<thead>
<tr>
<th>Previous Frame</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old Frame Ptr</td>
<td>secureData</td>
</tr>
<tr>
<td>...</td>
<td>buffer[0]</td>
</tr>
</tbody>
</table>
Control Flow Hijacking

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

![Diagram showing control flow hijacking and stack canaries]
Control Flow Hijacking

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

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

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

![Diagram showing control flow hijacking with stack canaries](image-url)
Control Flow Hijacking

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

Previous Frame

<table>
<thead>
<tr>
<th>Old Frame Ptr</th>
<th>secureData</th>
<th>buffer[15]</th>
<th>buffer[14]</th>
<th>...</th>
<th>buffer[0]</th>
</tr>
</thead>
</table>

Old Frame Ptr

| Canary        | secureData | buffer[15] | buffer[14] | ... | buffer[0] |

Abort on return 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:

```
<table>
<thead>
<tr>
<th>Previous Frame</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<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>
```
Control Flow Hijacking

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

```
Previous Frame
  Return Address
    Canary
    Old Frame Ptr
    secureData
    buffer[15]
    buffer[14]
    ...
    buffer[0]
```

shell code:  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⊕X

```
Previous Frame
Return Address
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]
```

```
Fake Argument
Ptr To Function
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]
```

```
"/usr/bin/minesweeper"
```
Return to libc Attacks

- Reuse existing code to bypass \(W \oplus X\)

```
Previous Frame
Return Address
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]
```

```
Fake Argument
Ptr To Function
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]
```

```
"/usr/bin/minesweeper"
```

Returning to common library code still works.
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>Old Frame Ptr</td>
</tr>
<tr>
<td>Old Frame Ptr</td>
<td>secureData</td>
</tr>
<tr>
<td>secureData</td>
<td>buffer[15]</td>
</tr>
<tr>
<td>buffer[14]</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>buffer[0]</td>
</tr>
</tbody>
</table>

Fake Argument:
- Old Frame Ptr
- secureData
- buffer[15]
- buffer[14]
- ...
- buffer[0]

“/usr/bin/minesweeper”

system()

Returning to common library code still works.

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 $W \oplus X$
- Return Oriented Programming
  - Build new functionality from pieces of existing functions
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 $W \oplus X$

- Return Oriented Programming
  - Build new functionality from pieces of existing functions

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

void b() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}

void c() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}

void d() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}

void e() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}

void f() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}

void g() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}

void h() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
```
Return to libc Attacks

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

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

NCurses

Stack

Heap

LibC

Program

Run 1
ASLR

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

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

Run 1

- NCurses
- Stack
- Heap
- LibC
- Program

Run 2

- Stack
- Heap
- LibC
- NCurses
- Program

But even this is "easily" broken
### ASLR

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

<table>
<thead>
<tr>
<th>NCurses</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Heap</td>
</tr>
<tr>
<td></td>
<td>LibC</td>
</tr>
<tr>
<td></td>
<td>Program</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>LibC</td>
</tr>
<tr>
<td>NCurses</td>
</tr>
<tr>
<td>Program</td>
</tr>
</tbody>
</table>

---

But even this is "easily" broken

Just leak a pointer first...
Mitigations

- Several automated *mitigations* are available
  - Approaches for lessening the likelihood & impact of a vulnerability
Mitigations

- Several automated *mitigations* are available
  - Approaches for lessening the likelihood & impact of a vulnerability

- **How can you prevent the core vulnerabilities we have discussed so far?**
  - Are there common points you can break? (Point in a kill chain)
Mitigations

- Several automated *mitigations* are available
  - Approaches for lessening the likelihood & impact of a vulnerability

- How can you prevent the core vulnerabilities we have discussed so far?
  - Are there common points you can break? (Point in a kill chain)

- *Are there obvious limitations with these techniques?*
Control Flow Integrity

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

```c
foo = ... 
foo();
```
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

- What problem from context sensitivity reappears for returns?
Control Flow Integrity

- What problem from context sensitivity reappears for returns?

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

- What problem from context sensitivity reappears for returns?
Control Flow Integrity

- What problem from context sensitivity reappears for returns?

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

- What problem from context sensitivity reappears for returns?

Can disambiguate call site/return pairs with a shadow stack
Control Flow Integrity

- What problem from context sensitivity reappears for returns?

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

Can disambiguate call site/return pairs with a shadow stack

```bash
clang -fsanitize=cfi -fsanitize=safe-stack
```
Control Flow *Bending*

- Suppose we have perfect CFI; how good is it?
Control Flow *Bending*

- Suppose we have perfect CFI; how good is it?
  - Can we skip using stacks?
Control Flow *Bending*

- Suppose we have perfect CFI; how good is it?
  - Can we skip using stacks?

  No!
Control Flow Bending

- Suppose we have perfect CFI; how good is it?
  - Can we skip using stacks?

Why might CFI offer porous defenses?
Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?
Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?
  - Coarse static approximations.
  - Open up too many opportunities for attack.
Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?
  - Coarse static approximations.
  - Open up too many opportunities for attack.

- **Fully precise CFI**
  - Include only those edges necessary for the dynamic correctness of the program.
  - Undecidable in general
Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?
  - Coarse static approximations.
  - Open up too many opportunities for attack.

- **Fully precise CFI**
  - Include only those edges necessary for the dynamic correctness of the program.
  - Undecidable in general

If fully precise CFI is broken, then CFI is broken.
Dispatcher Functions

- Some key functions open up substantial opportunities, e.g. memcpy
Dispatcher Functions

- Some key functions open up substantial opportunities, e.g. memcpy
- Any function that calls them becomes an attack surface
What Does CFI+Shadow Stacks Give?

- No longer able to do ROP
What Does CFI+Shadow Stacks Give?

- No longer able to do ROP

Arbitrary ROP gadgets are broken.
What Does CFI+Shadow Stacks Give?

- No longer able to do ROP
- Still able to do return to libc!
What Does CFI+Shadow Stacks Give?

- No longer able to do ROP
- Still able to do return to libc!
- Worse: `printf` alone provides a Turing complete attack surface
The trend going forward

Root cause of CVEs by patch year

[Graphic showing trend of CVEs by patch year]

[Matt Miller – BlueHat 2019]
Data Oriented Programming

- It only gets worse.
Security in Process & Design
Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation
  - Detection & Response
Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation
  - Detection & Response

Countermeasures
Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation
  - Detection & Response

Considering only one aspect is insufficient
Integrating Security into Development

- Managing security issues requires considering:
  - Prevention
  - Mitigation
  - Detection & Response

- Managing security within the development process is challenging:
  - Often poorly incentivized
  - Many do not possess required knowledge
  - Ownership of the problem is passed around
  - Many teams assume it does not even matter

Countermeasures
Integrating Security into Development

- We have classic guidelines for secure design [Saltzer and Schroeder 1975] more recently we have guidelines for secure process
Integrating Security into Development

- We have classic guidelines for secure design [Saltzer and Schroeder 1975] more recently we have guidelines for secure process
  - Microsoft's SDL
  - OWASP
  - BSIMM
Integrating Security into Development

- We have classic guidelines for secure design [Saltzer and Schroeder 1975] more recently we have guidelines for secure process
  - Microsoft’s SDL
  - OWASP
  - BSIMM

- Each approach provides recommendations for actions and feedback within the SDLC
Integrating Security into Development

- We have classic guidelines for secure design [Saltzer and Schroeder 1975] more recently we have guidelines for secure process
  - Microsoft’s SDL
  - OWASP
  - BSIMM

- Each approach provides recommendations for actions and feedback within the SDLC

We will explicitly consider process then design. There is some redundancy.
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) **Include security in testing**
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) **Perform code analysis**
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) **Perform code review for security**
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) **Perform post-development testing**
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Perform secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications
Managing Security in the SDLC

• Why do teams succeed or fail? [Assal & Chiasson, 2018]
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning
Managing Security in the SDLC

- **Why do teams succeed or fail?** [Assal & Chiasson, 2018]
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning
Managing Security in the SDLC

- **Why do teams succeed or fail?** [Assal & Chiasson, 2018]
  1) Division of labour
  2) **Security knowledge**
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning
Managing Security in the SDLC

- Why do teams succeed or fail? [Assal & Chiasson, 2018]
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning
Managing Security in the SDLC

- Why do teams succeed or fail? [Assal & Chiasson, 2018]
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning
Managing Security in the SDLC

- Why do teams succeed or fail? [Assal & Chiasson, 2018]
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning
Managing Security in the SDLC

- **Why do teams succeed or fail?** [Assal & Chiasson, 2018]
  1. Division of labour
  2. Security knowledge
  3. Company culture
  4. Resource availability
  5. External pressure
  6. Experiencing failure and learning
Designing for Security

- Half of security issues are design problems. [McGraw 2006]
Designing for Security

- Half of security issues are design problems. [McGraw 2006]

- Secure designs manage threats to CIA properties.
  - Threat modeling needs to be one of the first steps as in SDLC guidelines
Designing for Security

- Half of security issues are design problems. [McGraw 2006]
- **Secure designs manage threats to CIA properties.**
  - Threat modeling needs to be one of the first steps as in SDLC guidelines
  - Too weak – you won’t defend against the threats you need to
  - Too strong – you’ll waste resources defending against phantoms
Designing for Security

- Half of security issues are design problems. [McGraw 2006]

- Secure designs manage threats to CIA properties.
  - Threat modeling needs to be one of the first steps as in SDLC guidelines
  - Too weak – you won’t defend against the threats you need to
  - Too strong – you’ll waste resources defending against phantoms
  - Define realistic threat models (e.g. using STRIDE or more recent approaches)
Designing for Security

• Key principles from Saltzer & Schroeder
  – Economy of mechanism – keep things simple for easy inspection
  – Fail safe defaults – require permission rather than exclusion
  – Complete mediation – every access of every object should check authority
  – Open design – no security through obscurity
  – Separation of privilege – e.g. prefer two key locks
  – Least privilege – each actor should have fewest privileges necessary for a job
  – Least common mechanism – avoid shared mechanisms (single PoF & channel)
  – Psychological acceptability – make policies that people will use
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - **Fail safe defaults** – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - **Complete mediation** – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - **Open design – no security through obscurity**
  - Separation of privilege – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - **Separation of privilege** – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – e.g. prefer two key locks
  - **Least privilege** – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use
• **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - **Least common mechanism** – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - **Psychological acceptability** – make policies that people will use
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

This turns out to be exceedingly challenging. Usable security has been a growing area.
Designing for Security

- **Key principles from Saltzer & Schroeder**
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – e.g. prefer two key locks
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

- **Pfleeger & Lawrence**
  - Easiest penetration, weakest link, adequate protection, & effectiveness
Designing for Security

- What might a good design look like in practice?
  - Let us consider developing a mail client.
Future Directions

- Automating isolation guarantees in adversarial environments
- Making privilege specification & management easier