CMPT 473
Software Quality Assurance

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 Possibly 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]
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]

\[ \Delta \text{Data} \rightarrow \text{Dangling or OOB} \rightarrow \text{Read or Write} \]
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]

\[\begin{array}{c}
\text{Dangling or OOB }^* \\
\downarrow \quad \Delta \text{ Code} \\
\text{Code Corruption}
\end{array}\]
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]
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]
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

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

Previous Frame

Addresses

0x000

Stack

0xFFF
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);
}

Stack frame for foo

Addresses

Stack

Previous Frame

Return Address

Old Frame Ptr

secureData

buffer[15]

buffer[14]

...  

buffer[0]
Data Only Attacks

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

```c
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

![Diagram showing control flow hijacking with stack canaries](image_url)
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⊕X
Control Flow Hijacking

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

shell code:

- Previous Frame
- Return Address
- Canary
- Old Frame Ptr
- secureData
- buffer[15]
- buffer[14]
- ...
- buffer[0]
Control Flow Hijacking

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

shell code:

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

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"
system()
```
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”

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
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
Return to libc Attacks

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

Run 1

NCurses
Stack
Heap
LibC
Program

Run 2

Stack
Heap
LibC
NCurses
Program

But even this is “easily” broken
Control Flow Integrity

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

```
foo = ...
foo();
```
Control Flow Integrity

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

```cpp
foo = ...
if foo not in [...] abort()
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() {
...
...
...
...
...
...
...
}
```

`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
  - ...

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?
SQL Injection


- 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?

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
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?
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 naiveté 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