CSE543 - Introduction to Computer and Network Security

Module: Software Vulnerabilities

Professor Trent Jaeger
Programming

• Why do we write programs?
  ‣ Function

• What functions do we enable via our programs?
  ‣ Some we want -- some we don’t need
  ‣ Adversaries take advantage of such “hidden” function
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);

What if packet is larger than 1000 bytes?
Address Space Layout

- **Stack**
  - Can write the stack vars without limits

- **Type Safety**
  - Can write outside a data structure using its reference

- **Execution Integrity**
  - Can jump to arbitrary points in the program
    - Function pointers
    - Return addresses
Buffer Overflow

• How it works
Buffer Overflow Defense

- "Canary" on the stack
  - Random value placed between the local vars and the return address
  - If canary is modified, program is stopped

- Alternative:
  - How does this prevent return-to-libc?

- Are we done?
A Simple Program

```c
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}
if (authenticated)
    ProcessPacket(packet);
```

What if packet is only 1004 bytes?
Overflow of Local Variables

• Don’t need to modify return address
  ‣ Local variables may affect control

• What kinds of local variables would impact control?
  ‣ Ones used in conditionals (example)
  ‣ Function pointers

• What can you do to prevent that?
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);
Heap Overflow

• Overflows may occur on the heap also
  ‣ Heap has data regions and metadata

• Attack
  ‣ Write over heap with target address (heap spraying)
  ‣ Hope that victim uses an overwritten function pointer before program crashes
Another Simple Program

```c
int size = BASE_SIZE;
char *packet = (char *)malloc(1000);
char *buf = (char *)malloc(1000+BASE_SIZE);

strcpy(buf, FILE_PREFIX);
size += PacketRead(packet);
if ( size < sizeof(buf)) {
    strcat(buf, packet);
    fd = open(buf);
}
```

Any problem with this conditional check?
Integer Overflow

• Signed variables represent positive and negative values
  ‣ Consider an 8-bit integer: -128 to 127
  ‣ Weird math: $127 + 1 = ???$

• This results in some strange behaviors
  ‣ $\text{size} += \text{PacketRead}(\text{packet})$
    • What is the possible value of size?
  ‣ $\text{if ( size < sizeof(buf)) }$
    • What is the possible result of this condition?

• How do we prevent these errors?
A Simple Program

```c
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessQuery("Select", partof(packet));
```

Any problem with this query request?
Parsing Errors

• Have to be sure that user input can only be used for expected function
  ‣ SQL injection: user provides a substring for an SQL query that changes the query entirely (e.g., add SQL operations to query processing)

```
SELECT fieldlist
  FROM table
WHERE field = 'anything' OR 'x'='x';
```

• Goal: format all user input into expected types and ranges of values
  ‣ Integers within range
  ‣ Strings with expected punctuation, range of values

• Many scripting languages convert data between types automatically -- are not type-safe -- so must be extra careful
Character Strings

• String formats
  ‣ Unicode
    • ASCII -- 0x00 -- 0x7F
    • non-ASCII -- 0x80 -- 0xF7
    • Also, multi-byte formats
  ‣ Decoding is a challenge
    • URL: [IPaddr]/scripts/%c0%af../winnt/system32
    • Decodes to /winnt/system32
  ‣ Markus Kuhn’s page on Unicode resources for Linux
    • www.cl.cam.ac.uk/~mgk25/unicode.html
Secure Programming

- David Wheeler’s Secure Programming for Linux and UNIX
  - Validate all input; Only execute application-defined inputs!
  - Avoid buffer overflows (and other code injection problems)
  - Minimize process privileges
  - Carefully invoke other resources
  - Send information back carefully

Diagram:

- Bad
- Server
- Worker

- Validate Input
- Avoid Overflows
- Minimize Privilege
- Invoke Safely
- Return little
Name Resolution

- Processes often use names to obtain access to system resources.

- A nameserver (e.g., OS) performs name resolution using namespace bindings (e.g., directory) to convert a name (e.g., filename) into a system resource (e.g., file).
  - Filesystem, System V IPC, ...

```
P open("/var/mail/root")
```

- Namespace (filesystem)

```
Name (filename)
```

```
Bindings (directories)
```

```
Resource (file)
```

Sunday, September 30, 12
Attacks on Name Resolution

• Improper Resource Attack
  ‣ Adversary controls final resource in unexpected ways
  ‣ Untrusted search paths (e.g., Trojan library), file squatting
  ‣ Victim expects high integrity, gets low integrity instead
• Improper Resource Attack
  ‣ Adversary **controls final resource** in unexpected ways
  ‣ Untrusted search paths (e.g., Trojan library), file squatting
  ‣ Victim expects high integrity, gets low integrity instead

```
V_{root} \xrightarrow{\text{open("/var/mail/root")}} \text{/} \xrightarrow{} \text{var} \xrightarrow{} \text{mail} \xrightarrow{} \text{root}
```

A_{mail}
Attacks on Name Resolution

- Improper Binding Attack
  - Adversary controls bindings to redirect victim to a resource not under adversary’s control (confused deputy)
  - Symbolic link, hard link attacks
  - Victim expects low integrity/secrecy, gets high instead

```
V_{root} \rightarrow \text{open("/var/mail/root")}
```

```
/ \rightarrow \text{var} \rightarrow \text{mail} \rightarrow \text{root}
```

```
A_{mail} \rightarrow \text{etc} \rightarrow \text{passwd}
```

Link
Attacks on Name Resolution

- Race Conditions
  - Adversary exploits non-atomicity in “check” and “use” of resource to conduct improper resource and improper binding attacks
  - Well-known “TOCTTOU” attacks
Attacks on Name Resolution

• Race Conditions
  ‣ Adversary exploits non-atomicity in “check” and “use” of resource to conduct improper resource and improper binding attacks
  ‣ Well-known “TOCTTOU” attacks
Attacks on Name Resolution

• Race Conditions
  ‣ Adversary exploits non-atomicity in “check” and “use” of resource to conduct improper resource and improper binding attacks
  ‣ Well-known “TOCTTOU” attacks
How Serious a Problem?

• Name resolution vulnerabilities accounts for 5-10% CVE entries each year

• These are particularly hard to eradicate as they involve multiple parties
  ‣ Programmers who write code
    ```c
    if ((stat("/var/mail/root", st)) == 0 && !S_ISLNK(st->st_mode))
    ```
  ‣ OS distributors who define access control policies
    ```bash
    drwxrwsr-x  2 root  mail  4096 2011-01-13 10:06 mail
    ```
  ‣ Administrators who configure end system
Difficult to Prevent

- Manual checks can easily overlook vulnerabilities
- But, misses already existing file squat!

```c
01 /* filename = /var/mail/root */
02 /* First, check if file already exists */
03 fd = open (filename, flg);
04 if (fd == -1) {
05     /* Create the file */
06         fd = open(filename, O_CREAT|O_EXCL);
07         if (fd < 0) {
08             return errno;
09         }
10 }
11 /* We now have a file. Make sure we did not open a symlink. */
12 struct stat fdbuf, filebuf;
13 if (fstat (fd, &fdbuf) == -1)
14     return errno;
15 if (lstat (filename, &filebuf) == -1)
16     return errno;
17 /* Now check if file and fd reference the same file, file only has one link, file is plain file. */
18 if ((fdbuf.st_dev != filebuf.st_dev || fdbuf.st_ino != filebuf.st_ino || fdbuf.st_nlink != 1 || filebuf.st_nlink != 1 || (fdbuf.st_mode & S_IFMT) != S_IFREG)) {
19     error (_("%s must be a plain file with one link"), filename);
20     close (fd);
21     return EINVVAL;
22 }
23 /* If we get here, all checks passed. */
24 read(fd, ...)
```
Fundamental Problem

- Security problems occur because low-integrity adversary processes share the same OS namespaces as high-integrity victim processes
  - Adversary processes attempt to affect name resolution of victim processes
- Permissions for `/var/mail`
  - Group mail can create and delete files

```
drwxrwsr-x 2 root mail 4096 2011-01-13 10:06 mail
```
STING Approach

• Thus, we have to **actively change** the namespace to create adversarial scenarios
  ‣ And evaluate process response to scenario

• We take inspiration from “grey-box” testing
  ‣ Feed known adversarial inputs to programs and examine process response (e.g., detect SQL injection vulnerability)

```
Generate Adversarial Input

'test'; drop table name;

V

Vulnerable!

db.exec('drop table name');
```

Study Program Response
Adversary Accessibility

• Under DAC adversary model
  ‣ Only 4% (Fedora) and 5.7% (Ubuntu) of total name resolution entrypoints were accessible to adversaries
  ‣ Only 0.3% (Fedora) and 0.9% (Ubuntu) of total name resolutions were vulnerable

<table>
<thead>
<tr>
<th>Adversary model</th>
<th>Total Resolutions</th>
<th>Adversary Access</th>
<th>Vulnerable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAC - Ubuntu</td>
<td>2345</td>
<td>134 (5.7%)</td>
<td>21 (0.9%)</td>
</tr>
<tr>
<td>DAC - Fedora</td>
<td>1654</td>
<td>66 (4%)</td>
<td>5 (0.3%)</td>
</tr>
</tbody>
</table>
Vulnerabilities Found

<table>
<thead>
<tr>
<th>Program</th>
<th>Vuln. Entry</th>
<th>Priv. Escalation DAC: uid-&gt;uid</th>
<th>Distribution</th>
<th>Previously known</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbus-daemon</td>
<td>2</td>
<td>messagebus-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>landscape</td>
<td>4</td>
<td>landscape-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>Startup scripts (3)</td>
<td>4</td>
<td>various-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>mysql</td>
<td>2</td>
<td>mysql-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>mysql_upgrade</td>
<td>1</td>
<td>mysql-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>tomcat script</td>
<td>2</td>
<td>tomcat6-&gt;root</td>
<td>Ubuntu</td>
<td>Known</td>
</tr>
<tr>
<td>lightdm</td>
<td>1</td>
<td>*-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>bluetooth-applet</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>java (openjdk)</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Both</td>
<td>Known</td>
</tr>
<tr>
<td>zeitgeist-daemon</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Both</td>
<td>Unknown</td>
</tr>
<tr>
<td>mountall</td>
<td>1</td>
<td>*-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>mailutils</td>
<td>1</td>
<td>mail-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>bsd-mailx</td>
<td>1</td>
<td>mail-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>cupsd</td>
<td>1</td>
<td>cups-&gt;root</td>
<td>Fedora</td>
<td>Known</td>
</tr>
<tr>
<td>abrt-server</td>
<td>1</td>
<td>abrt-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>yum</td>
<td>1</td>
<td>sync-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>x2gostartagent</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Extra</td>
<td>Unknown</td>
</tr>
<tr>
<td><strong>19 Programs</strong></td>
<td><strong>26</strong></td>
<td></td>
<td></td>
<td><strong>21 Unknown</strong></td>
</tr>
</tbody>
</table>

Table 4: Table showing the total number of distinct entrypoints.

Table 6.1.1 Finding Vulnerabilities

We found 26 vulnerabilities in Ubuntu-specific scripts. For example, CVE-2011-4406 and CVE-2011-3151 were assigned to two other users.

Interestingly, we note that 6 vulnerabilities of all types, including 7 that required race conditions, were found. Interestingly, we note that 6 vulnerabilities of all types, including 7 that required race conditions, were found.

A single entrypoint may be vulnerable to more than one kind of attack. We note that S even if a network daemon running as root gets compromised, it should still not compromise the whole system arbitrarily. However, we found that the SELinux policy allowed subjects we consider untrusted to carry out a further attack on the PostgreSQL daemon.

This can be achieved, for example, by remote network users.

We thus believe that the programs in Table 6.1.1 are not only vulnerable to attacks but also latent attacks. This is the number of TOCTTOU vulnerabilities, where a check is made but the use is improper.
Take Away

• Programs have function
  ‣ Adversaries can exploit unexpected functions

• Vulnerabilities due to malicious input
  ‣ Buffer overflows
  ‣ Heap overflows
  ‣ Injection attacks

• Is an integer overflow possible in type-safe languages?
  ‣ E.g., Java

• Vulnerabilities due to controlling name resolution
  ‣ Improper resource: file squatting, untrusted search path
  ‣ Improper bindings: link traversal and TOCTTOU