Advanced System Security: Vulnerabilities

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Program Vulnerabilities

• What are properties of a program vulnerability?
  ‣ Program flaw
  ‣ Accessible to adversary
  ‣ Adversary is capable of exploiting
int authenticated = 0;
char packet[1000];

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

if (authenticated)
    ProcessPacket(packet);
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 larger than 1000 bytes?
Address Space Layout

- Write beyond variable limit
  - Can write the without limits in some languages
- Can impact values
  - In heap, on stack, in data
- Can impact execution integrity
  - Can jump to arbitrary points in the program
  - Function pointers
  - Return addresses
Buffer Overflow

• How it works
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
A Simple Program

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

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

What if we allocate the packet buffer on the heap?
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
  ‣ `size += PacketRead(packet)`
    • What is the possible value of `size`?
  ‣ `if ( size < sizeof(buf)) {}`
    • What is the possible result of this condition?

• How can an adversary leverage this?
A Simple Program

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)

```sql
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 Input Handling

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

Diagram:
- Server
  - Validate Input
  - Avoid Overflows
  - Minimize Privilege
  - Invoke Safely
- Bad
  - Return little
- Worker
Program Vulnerabilities

• Whose fault are these flaws?
  ‣ Programmer
  ‣ OS distributor
  ‣ System administrator
Other Vulnerabilities

• These may also be the programmers’ fault
  ‣ Missing an authorization or authentication check
  ‣ Confused deputy
  ‣ Integrate untrustworthy code
  ‣ Logic errors
  ‣ Vet the provenance of command line arguments and environment variables

• But, the security policy provide accessibility and enables exploitation
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, …

![Diagram showing name resolution process]

- **Name** (filename)
- **Bindings** (directories)
- **Resource** (file)
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
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

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

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

```mail
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

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

```bash
V_{\text{root}} \xrightarrow{\text{lstat("/var/mail/root")}} / \rightarrow \text{var} \rightarrow \text{mail} \rightarrow \text{root}
```

```bash
\text{A}_{\text{mail}} \xrightarrow{} \text{etc} \rightarrow \text{passwd}
```
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

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

```
A_{mail} \xrightarrow{\text{Link}} etc \xrightarrow{\text{Link}} passwd
```

```
\xrightarrow{\text{Link}} var \xrightarrow{\text{Link}} mail \xrightarrow{\text{Link}} root
```
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
19     || fdbuf.st_ino != filebuf.st_ino
20     || fdbuf.st_nlink != 1
21     || filebuf.st_nlink != 1
22     || (fdbuf.st_mode & S_IFMT) != S_IFREG)) {
23     error ("%s must be a plain file with one link", filename);
24 }
25     close (fd);
26     return EINVAL;
27 }
28 /* If we get here, all checks passed. */
29     read(fd, ...)
```

Squat during create

Symbolic link

Hard link, race conditions
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

```bash
drwxrwsr-x 2 root mail 4096 2011-01-13 10:06 mail
```
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
    ```
    drwxrwsr-x  2 root mail  4096 2011-01-13 10:06 mail
    ```
  ‣ Administrators who configure end system
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>
Take Away

• Vulnerabilities occur when
  ‣ A program flaw
  ‣ Is accessible to an adversary
  ‣ Who can exploit it

• Some vulnerabilities are caused by programmer errors
  ‣ But if compiler passes it is it a programmer error?
  ‣ And some logic errors are difficult to tease out

• Adversary accessibility is an important consideration
  ‣ Name resolution vulnerabilities are only possible in few cases
  ‣ Ambiguous, non-atomic APIs make vulnerabilities common in those cases