Program Security

CMPSC 443 - Spring 2012
Introduction Computer and Network Security
Professor Jaeger

www.cse.psu.edu/~tjaeger/cse443-s12/
Programming

• Why do we write programs?
  – Function

• What functions do we enable via our programs?
  – Often more than the programmer expects
  – 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 and Assumptions

- Depends on the layout of a computer process’s memory
  - Note that the stack grows downward

- Depends on lack of type safety in programming language
  - Can write outside a data structure using its reference

- Difference between logical model and implementation
  - Can’t assume everything works according to the logical model
Buffer Overflow

• How it works

![Stack Frame Diagram]

- Previous Function
- Func Parameters
- Return Address
- Local Var
- Buffer
- Local Var
- New Rtn
- Evil Code
- Evil Code
- Evil Code
- Evil Code

Stack Frame
A Simple Program

int authenticated = 0;
char packet[1000];

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

if (authenticated)
    ProcessPacket(packet);

How would you fix this problem?

Can we depend on programmers to prevent this problem?
Return-to-libc

- Possible defense: non-executable stack
- Possible defense: randomize stack base
- *Do we need to run code on the stack?*
Return-to-libc

- Possible defense: non-executable stack
- Possible defense: randomize stack base

- *Do we need to run code on the stack?*
  - Don’t need to inject code
  - There is already plenty of code

- Just fix stack as necessary to call an existing function
  - and jump to that address
  - Usually library functions (e.g., system) are used

- Can we really do very much with such an attack?
Return-to-libc

• **Bad news**: you can run any code via a generalization of return-to-libc called *return-oriented programming*
  – Turing-complete system
  – A universal computer

• So we need to work hard to prevent such attacks
### Buffer Overflow Defenses

- **“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?**

<table>
<thead>
<tr>
<th>Previous Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Func Parameters</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>CANARY</td>
</tr>
<tr>
<td>Local Var</td>
</tr>
<tr>
<td>Buffer</td>
</tr>
<tr>
<td>Local Var</td>
</tr>
</tbody>
</table>
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?
Buffer 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?
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);
```

How 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
Heap Overflow

• Defenses
  – Could use canaries on the stack, but difficult to make efficient
  – Alternative is to check space in the heap buffer before writing (HeapShield)
    • Can be done efficiently
    • But needs to be done for every function that writes to buffers
Memory Safety

• Are such attacks possible if
  – writing to string buffers is bounded or
  – string buffers are automatically expanded to store all the data?

• Example
  – When writing a 50 byte string to \textit{buf} either the write is limited by the size of character string \textit{buf} or
  – Memory allocated for character strings is automatically updated to hold all 50 bytes

• This is not the case in some languages that are not \textit{memory-safe} (e.g., C, C++)
  – Data updates occur by memory reference (pointers), so the format of data structures has no impact
    • Can write beyond the end of a data structure
Prevent Memory Safety Errors

• How would we prevent such errors fundamentally?
Prevent Memory Safety Errors

- How would we prevent such errors fundamentally?
  - Enforce memory safety (data-flow integrity)
  - Enforce stack execution semantics (control-flow integrity)
- We’ll discuss these later
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 \(\text{size}\)?
  – \(\text{if } (\text{size} < \text{sizeof(\text{buf})}) \{\)
    • What is the possible result of this condition?

• How do we prevent these errors?
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 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 showing the process of secure programming with Bad, Validate Input, Minimize Privilege, Avoid Overflows, Invoke Safely, and Worker. The diagram illustrates the flow from Bad to Validate Input to Server, and then to Worker, with arrows indicating the process steps.]
Take Away

• Programs have function
  – More than the programmer expects

• Variety of function possible depends on
  – Memory safety
  – Type safety
  – Accuracy of parsing

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

• Programmers must be aware of such problems and know how to fix them
  – This is difficult in general (but you should be able to prevent several flaws), so later we will discuss program analysis tools to detect (repair?) such flaws