Module: Program 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
A Simple Program

```c
void myfunc()
{
    char string[16];
    printf("Enter a string\n");
    scanf("%s", string);
    printf("You entered: %s\n", string);
}
int main()
{
    myfunc();
}
```
A Simple Program

```c
void myfunc()
{
    char string[16];
    printf("Enter a string\n");
    scanf("%s", string);
    printf("You entered: %s\n", string);
}

int main()
{
    myfunc();
}
```

root@newyork:~/test# ./a.out
Enter a string
mysting
You entered: mysting
void myfunc() {
    char string[16];
    printf("Enter a string\n");
    scanf("%s", string);
    printf("You entered: %s\n", string);
}

int main() {
    myfunc();
}

root@newyork:~:/test# ./a.out
Enter a string
mysting
You entered: mysting

root@newyork:~:/test# ./a.out
Enter a string
ajhsoieuhgesklджфгклжгжджфгслджфгсклжргфджк
You entered: ажпоёйжегслдждфкордждлджфгсклжргфджк
Segmentation fault (core dumped)
What Happened?

- Brief refresher on program address space
  - Stack -- local variables
  - Heap -- dynamically allocated (malloc, free)
  - Data -- global, uninitialized variables
  - Text -- program code

```bash
root@newyork:~:/test# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 0000
08053000-08054000 r--p 00000000 08:01 0000
08054000-08055000 rw-p 00000000 08:01 0000
08c20000-08c41000 rw-p 00000000 00:00 0
b7352000-b7552000 r--p 00000000 08:01 10346
b7552000-b7553000 rw-p 00000000 00:00 0
b7553000-b7700000 r-xp 00000000 08:01 122
b7700000-b7702000 r--p 00100000 08:01 122
b7702000-b7703000 rw-p 00100000 08:01 122
b7703000-b7706000 rw-p 00000000 00:00 0
b7706000-b770f000 rw-p 00000000 00:00 0
b770f000-b7710000 r-xp 00000000 00:00 0
b7710000-b7730000 r-xp 00000000 08:01 102
b7730000-b7731000 r--p 00000000 08:01 102
b7731000-b7732000 rw-p 00200000 08:01 102
bf0e2000-bf0e2000 rw-p 00000000 00:00 0
```

What Happened?

Stack

main() parameters (argc, argv)

return address

saved frame pointer

main() local vars

myfunc() parameters (void)

return address

saved frame pointer

myfunc() local vars

string[16]
What Happened?

Stack

- main() parameters (argc, argv)
- return address
- saved frame pointer
- main() local vars
- myfunc() parameters (void)
- sghfjdsh
- return address
- saved frame pointer
gjlkhgfd

- myfunc() local vars
- jlkseghrueloshja
- string[16]

Code:

```c
void my_func()
{
    char string[16];
    printf("Enter a string\n");
    scanf("%s", string);
    printf("You entered: %s\n", string);
}

int main(int argc, char *argv[])
{
    my_func();
    printf("Done");
}
```

CSE543 - Introduction to Computer and Network Security
Exploiting Buffer Overflow

Stack

- main() parameters (argc, argv)
- return address
- saved frame pointer
- main() local vars
- myfunc() parameters (void)
- return address
- saved frame pointer
- myfunc() local vars
  - string[16]
Exploiting Buffer Overflow

Stack

- main() parameters (argc, argv)
- return address
- saved frame pointer
- main() local vars
- myfunc() parameters (void)
- return address
- pc + 8
- saved frame pointer
- my evil code
- myfunc() local vars
- more evil code
- string[16]

void my_func()
{
    char string[16];
    printf("Enter a string\n");
    scanf("%s", string);
    printf("You entered: %s\n", string);
}

int main(int argc, char *argv[])
{
    my_func();
    printf("Done");
}

(libc)
_start:
setup
main();
cleanup
A First Defense

- What if we made the stack non-executable?
  - AMD NX-bit
  - More general: \( W \oplus X \)
- Attacker response
  - return-to-libc

```
myfunc() parameters (void)

return address

saved frame pointer

myfunc() local vars
string[16]
```

```
(int system(const char *command)
{
    ... 
});
```
A First Defense

- What if we made the stack non-executable?
  - AMD NX-bit
  - More general: W (xor) X

- Attacker response
  - return-to-libc
Defense #2 - Canary

- "Canary" on the stack
  - Random value placed between the local vars and the return address
  - If canary is modified, program is stopped
- Have we solved buffer overflows?

<table>
<thead>
<tr>
<th>main() parameters(argc, argv)</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved frame pointer</td>
<td></td>
</tr>
<tr>
<td>myfunc() local vars</td>
<td>return address</td>
</tr>
<tr>
<td>myfunc() parameters (void)</td>
<td></td>
</tr>
<tr>
<td>saved frame pointer</td>
<td></td>
</tr>
<tr>
<td>myfunc() local vars</td>
<td>string[16]</td>
</tr>
</tbody>
</table>
Defense #2 - Canary

<table>
<thead>
<tr>
<th>myfunc() parameters (void)</th>
<th>main() local vars</th>
<th>return address</th>
<th>saved frame pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() parameters (argc, argv)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **“Canary” on the stack**
  - Random value placed between the local vars and the return address
  - If canary is modified, program is stopped
- **Have we solved buffer overflows?**

- CANARY
- saved frame pointer
- myfunc() local vars
- string[16]
## Canary Shortcomings

<table>
<thead>
<tr>
<th>main() parameters(argc, argv)</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved frame pointer</td>
<td>main() local vars</td>
</tr>
<tr>
<td>myfunc() parameters (void)</td>
<td>return address</td>
</tr>
<tr>
<td>CANARY</td>
<td>saved frame pointer</td>
</tr>
<tr>
<td>myfunc() local vars</td>
<td>string[16]</td>
</tr>
</tbody>
</table>

- Other local variables?
- Frame pointers?
- Anything left unprotected on stack can be used to launch attacks
- Not possible to protect everything
  - Varargs
  - Structure members
  - Performance
int authenticated = 0;
char packet[1000];

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

- myfunc() parameters
- return address
- CANARY
- saved frame pointer
- int authenticated
- char packet[1000]
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?
More Stack Defenses

- Separate control information (return address) and data
  - Shadow stacks
    - Maintain a shadow stack; compare return addresses on return
    - If different, stop program
    - Problem: performance

- Randomization
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 Overflows

- Overflows on heap also possible

```c
char *packet = malloc(1000);
ptr[1000] = 'M';
```

- “Classical” heap overflow corrupts metadata
  - Heap metadata maintains chunk size, previous and next pointers, ...
  - Heap metadata is *inline* with heap data
    - And waits for heap management functions (*malloc*, *free*) to write corrupted metadata to target locations
Heap Overflows

- Heap allocators maintain a doubly-linked list of allocated and free chunks
- `malloc()` and `free()` modify this list

![Heap Overflows Diagram]

Heap Overflows

- `free()` removes a chunk from allocated list
  \[
  \text{chunk2->bk->fd} = \text{chunk2->fd} \\
  \text{chunk2->fd->bk} = \text{chunk2->bk}
  \]

- By overflowing `chunk2`, attacker controls \text{bk} and \text{fd}
  - Controls both \text{where} and \text{what} data is written!
  - Arbitrarily change memory (e.g., function pointers)
Heap Overflow Defenses

• Separate data and metadata
  ‣ e.g., OpenBSD’s allocator (Variation of PHKmalloc)

• Sanity checks during heap management
  free(chunk2) -->
  assert(chunk2->prev->next == chunk2)
  assert(chunk2->next->prev == chunk2)
  ‣ Added to GNU libc 2.3.5

• Randomization

• Q. What are analogous defenses for stack overflows?
Other Heap Attacks

- Heap spraying
  - Combat randomization by filling heap with allocated objects containing malicious code
  - Use another vulnerability to overwrite a function pointer to any heap address, hoping it points to a sprayed object
  - Heuristic defenses
    - e.g., NOZZLE: If heap data is like code, flag attack
- Dangling pointers
- Memory reuse
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 < 1000+BASE_SIZE) {
    strcat(buf, packet);
    fd = open(buf);
}
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 \( \text{size} \)?
  - \( \text{if} ( \text{size} < 1000+\text{BASE\_SIZE} ) \) {
    - What is the possible result of this condition?
- How do we prevent these errors?
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 < 1000+BASE_SIZE) {
    strcat(buf, packet);
    fd = open(buf);
    printf(packet);
}

Any problem with this printf?
Format String Vulnerability

• Attacker control of the format string results in a format string vulnerability
  ‣ printf is a very versatile function
    • %s - dereferences (crash program)
    • %x - print addresses (leak addresses, break ASLR)
    • %n - write to address (arbitrarily change memory)
  
• Never use
  ‣ printf(string);

• Instead, use
  ‣ printf(“%s”, string);
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)

```sql
SELECT *
FROM students
WHERE student_name = 'Robert';
```

• 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 -- 0xBF
    • 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
  - Avoid the various overflows
  - Validate all input; Only execute application-defined inputs!
  - Minimize process privileges
  - Carefully invoke other resources
  - Send information back carefully

Diagram:
- Bad
  - Validate Input
  - Return little

- Server
  - Avoid Overflows
  - Minimize Privilege

- Worker
  - Invoke Safely
Take Away

• Programs have function
  ‣ Adversaries can exploit unexpected functions

• Vulnerabilities due to malicious input
  ‣ Low-level
    • Buffer, heap, integer overflows, format string vulnerabilities
  ‣ Injection attacks
    • Application-dependent

• If applicable, write programs in languages that eliminate vulnerabilities
  ‣ E.g., Type-safe languages such as Java