CSE543 - Introduction to Computer and Network Security

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
Broad Attack Categories

- Control-flow Attacks
  - Adversary directs program control-flow
    - E.g., return address overwrite through buffer overflow

- Data Attacks
  - Adversary exploits flaw to read/modify unexpected data
    - E.g., critical variable overwrite through buffer overflow

- Code Injection Attacks
  - Adversary tricks the program into executing their input
    - E.g., SQL injection attacks

- Other types of attacks on unauthorized access (later)
- See CWE (http://cwe.mitre.org/)
Memory Errors

• Control and data attacks are possible because some programming languages allow memory errors
  ‣ C and C++ for example

• A memory error occurs when the program allows an access to a variable to read/write to memory beyond what is allocated to that variable
  ‣ E.g., read/write beyond the end of a string
  ‣ Access memory next to the string

• Memory errors may be exploited to change the program’s control-flow or data-flow or to injection code
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
mystring
You entered: mystring
```
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
mystring
You entered: mystring
```

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

```
root@newyork:~:/test# cat /proc/self/maps
0048000-0053000 r-xp 00000000 08:01 131088 /bin/cat
0053000-0054000 r--p 00000000 08:01 131088 /bin/cat
0054000-0055000 rw-p 00000000 08:01 131088 /bin/cat
001b0000-001d000 rw-p 00000000 00:00 0 [heap]
b752000-b7552000 r--p 00000000 08:01 10346 /usr/lib/locale/locale-archive
b7552000-b7553000 rw-p 00000000 00:00 0
b7553000-b7700000 r-xp 00000000 08:01 122 /lib/i386-linux-gnu/libc-2.17.so
b7700000-b7720000 r--p 00100000 08:01 122 /lib/i386-linux-gnu/libc-2.17.so
b7720000-b7730000 rw-p 00100000 08:01 122 /lib/i386-linux-gnu/libc-2.17.so
b7730000-b7760000 rw-p 00000000 00:00 0
b7760000-b77f0000 rw-p 00000000 00:00 0 [vdso]
b77f0000-b7810000 r-xp 00000000 00:00 0
b7710000-b7730000 r-xp 00000000 08:01 102 /lib/i386-linux-gnu/ld-2.17.so
b7730000-b7731000 r--p 00010000 08:01 102 /lib/i386-linux-gnu/ld-2.17.so
b7710000-b7730000 r--p 00020000 08:01 102 /lib/i386-linux-gnu/ld-2.17.so
bfe2000-bfeced000 rw-p 00000000 00:00 0 [stack]
```
What Happened?

```
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");
}
```

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)
- return address
- saved frame pointer
- myfunc() local vars
- string[16]

Code snippet:
```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");
    (int) _start;
    setup
    main();
    cleanup
```
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]

```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");
}
```

(libc)

_start:
    setup
    main();
    cleanup
Exploiting Buffer Overflow

Stack

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

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
Prevent Code Injection
Prevent Code Injection

• What if we made the stack non-executable?
  ‣ AMD NX-bit
  ‣ More general: $W \text{xor} X$ (DEP in Windows)
Prevent Code Injection

- What if we made the stack non-executable?
  - AMD NX-bit
  - More general: W (xor) X (DEP in Windows)
Prevent Code Injection

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

```
myfunc() parameters (void)
return address
saved frame pointer
myfunc() local vars
string[16]
```
Prevent Code Injection

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

myfunc() parameters (void)

return address
saved frame pointer

myfunc() local vars
string[16]
Prevent Code Injection

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

```
myfunc() parameters (void)

pc of libc call()

saved frame pointer

arguments for libc call

myfunc() local vars

string[16]

(libc)

int system(const char *command)
{
    ...
}
```
Protect the Return Address

- **“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>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
</tr>
<tr>
<td>saved frame pointer</td>
</tr>
<tr>
<td>main() local vars</td>
</tr>
<tr>
<td>myfunc() parameters (void)</td>
</tr>
<tr>
<td>return address</td>
</tr>
</tbody>
</table>

| saved frame pointer           |
| myfunc() local vars string[16]|
Protect the Return Address

- "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>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
</tr>
<tr>
<td>saved frame pointer</td>
</tr>
<tr>
<td>main() local vars</td>
</tr>
<tr>
<td>myfunc() parameters (void)</td>
</tr>
<tr>
<td>return address</td>
</tr>
</tbody>
</table>

[CANARY]

| saved frame pointer           |
| myfunc() local vars           |
| string[16]                    |
Canary Shortcomings

- 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

### Variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Local Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>argc, argv</td>
<td>main() local vars</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return address</td>
</tr>
<tr>
<td>myfunc()</td>
<td>(void)</td>
<td>myfunc() local vars</td>
</tr>
<tr>
<td></td>
<td></td>
<td>saved frame pointer</td>
</tr>
<tr>
<td></td>
<td>string[16]</td>
<td></td>
</tr>
</tbody>
</table>
## Canary Shortcomings

<table>
<thead>
<tr>
<th>main() parameters(argc, argv)</th>
<th>• Other local variables?</th>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
<td>• Frame pointers?</td>
</tr>
<tr>
<td>saved frame pointer</td>
<td>• Anything left unprotected on stack can be used to launch attacks</td>
</tr>
<tr>
<td>main() local vars</td>
<td>• Not possible to protect everything</td>
</tr>
<tr>
<td>myfunc() parameters (void)</td>
<td>• Varargs</td>
</tr>
<tr>
<td>return address</td>
<td>• Structure members</td>
</tr>
<tr>
<td>CANARY</td>
<td>• Performance</td>
</tr>
<tr>
<td>saved frame pointer</td>
<td></td>
</tr>
<tr>
<td>myfunc() local vars</td>
<td></td>
</tr>
<tr>
<td>string[16]</td>
<td></td>
</tr>
</tbody>
</table>
A Simple Program

```c
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 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?
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

http://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf
Heap Overflows

\[ \text{chunk2->bk->fd} = \text{chunk2->fd} \]
\[ \text{chunk2->fd->bk} = \text{chunk2->bk} \]
Heap Overflows

- `free()` removes a chunk from allocated list

\[
\begin{align*}
\text{chunk2->bk->fd} &= \text{chunk2->fd} \\
\text{chunk2->fd->bk} &= \text{chunk2->bk}
\end{align*}
\]
Heap Overflows

- free() removes a chunk from allocated list

\[
\text{chunk2->bk->fd} = \text{chunk2->fd} \\
\text{chunk2->fd->bk} = \text{chunk2->bk}
\]
Heap Overflows

- free() removes a chunk from allocated list

\[ \text{chunk2->bk->fd} = \text{chunk2->fd} \]
\[ \text{chunk2->fd->bk} = \text{chunk2->bk} \]
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 \textit{where} and \textit{what} data is written!
  - Arbitrarily change memory (e.g., function pointers)

\[
\begin{align*}
\text{chunks1, 2, and 3 are joined by a doubly-linked list} \\
\text{chunk2 may be unlinked by rewriting 2 pointers} \\
\text{chunk2 is now unlinked}
\end{align*}
\]
Heap Overflows

• By overflowing chunk2, attacker controls \( bk \) and \( fd \)
  ‣ Controls both where and what data is written!
    • Assign chunk2->fd to value to want to write
    • Assign chunk2->bk to address X (where you want to write)
      • Less an offset of the \( fd \) field in the structure

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

• What’s the result?
Heap Overflows

• By overflowing chunk2, attacker controls bk and fd
  ‣ Controls both where and what data is written!
    • Assign chunk2->fd to value to want to write
    • Assign chunk2->bk to address X (where you want to write)
      • Less an offset of the fd field in the structure
  • Free() removes a chunk from allocated list
    \[ \text{chunk2->bk->fd} = \text{chunk2->fd} \]
    \[ \text{addrX->fd} = \text{value} \]
    \[ \text{chunk2->fd->bk} = \text{chunk2->bk} \]
    \[ \text{value->bk} = \text{addrX} \]

• What’s the result?
  • Change a memory address to a new pointer value (in data)
Heap Overflow Defenses

• Separate data and metadata
  ‣ e.g., OpenBSD’s allocator (Variation of PHKmalloc)
• Sanity checks during heap management

\[
\text{free}(\text{chunk2}) \rightarrow
\]

\[
\text{assert}(\text{chunk2} \rightarrow \text{fd} \rightarrow \text{bk} == \text{chunk2})
\]

\[
\text{assert}(\text{chunk2} \rightarrow \text{bk} \rightarrow \text{fd} == \text{chunk2})
\]

• Added to GNU \texttt{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

• Use-after-free
  ‣ Type confusion
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
- Use-after-free
  - Type confusion
Another Simple Program

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)) {
    return(-1)
} else
    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
  ‣ $size += \text{PacketRead}(\text{packet})$
    • What is the possible value of $size$?
  ‣ if ( $size >= 1000+BASE\_SIZE$ ) ... { 
    • What is the possible result of this condition?

• How do we prevent these errors?
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 < 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 -- 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
Take Away

- Programs have function
  - Adversaries can exploit unexpected functions
- Vulnerabilities due to malicious input
  - Subvert control-flow or critical data
    - Buffer, heap, integer overflows, format string vulnerabilities
  - Injection attacks
    - Application-dependent
- If applicable, write programs in languages that eliminate classes of vulnerabilities
  - E.g., Type-safe languages such as Java