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
Some 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

- Many 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 allow injection of code
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

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

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
ajhsoieurhgeskljdfghkljghsdjfhgsldkjfgghsklhrhgfdkj
You entered: ajhsoieurhgeskljdfghkljghsdjfhgsldkjfgghsklhrhgfdkj
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
08048000-08053000 r-xp 00000000 08:01 131088 /bin/cat
08053000-08054000 r--p 00000000 08:01 131088 /bin/cat
08054000-08055000 rw-p 00000000 08:01 131088 /bin/cat
08c20000-08c41000 rw-p 00000000 00:00 0 [heap]
b7352000-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-b7702000 r--p 001ad000 08:01 122 /lib/i386-linux-gnu/libc-2.17.so
b7702000-b7703000 rw-p 001af000 08:01 122 /lib/i386-linux-gnu/libc-2.17.so
b7703000-b7706000 rw-p 00000000 00:00 0
b770d000-b770f000 rw-p 00000000 00:00 0
b770f000-b7710000 r-xp 00000000 00:00 0 [vdso]
b7710000-b7730000 r-xp 00000000 08:01 102 /lib/i386-linux-gnu/ld-2.17.so
b7730000-b7731000 r--p 0001f000 08:01 102 /lib/i386-linux-gnu/ld-2.17.so
b7731000-b7732000 rw-p 00020000 08:01 102 /lib/i386-linux-gnu/ld-2.17.so
bfe02000-bfeced00 rw-p 00000000 00:00 0 [stack]
```

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:
```c
void myfunc()
{
    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");
}
```

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)

- address of string
- saved frame pointer
- more evil code

myfunc() local vars

- string[16]
- my evil code

CSE543 - Introduction to Computer and Network Security
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)

![Code Injection Diagram]

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]

(libc)
int system(const char *command)
{
    ...}
```
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

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

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

![Diagram showing heap allocation and deallocation with linked lists]

Heap Overflows

\[
\text{chunk2-}\rightarrow\text{bk-}\rightarrow\text{fd} = \text{chunk2-}\rightarrow\text{fd} \\
\text{chunk2-}\rightarrow\text{fd-}\rightarrow\text{bk} = \text{chunk2-}\rightarrow\text{bk}
\]
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

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

- 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 Overflows

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

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

• Address space randomization
  ‣ Make it difficult to predict where a particular program variable is stored in memory

• Rather than randomly locate every variable
  ‣ A simpler solution is to randomly offset each memory region

• Address space layout randomization (ASLR)
  ‣ Stack and heap are located at different base addresses each time the program is run
  ‣ NOTE: Always on a page offset, however, so limited in range of bits available for randomization

• Also, works for buffer 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
Heap Overflow Defenses

- Separate data and metadata
  - e.g., OpenBSD’s allocator (Variation of PHKmalloc)
- Sanity checks during heap management
  
  ```c
  free(chunk2) -->
  assert(chunk2->fd->bk == chunk2)
  assert(chunk2->bk->fd == chunk2)
  ```
  - Added to GNU libc 2.3.5
- Randomization
- Q. What are analogous defenses for stack overflows?
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
  ‣ $\text{size} += \text{PacketRead}($packet$)$
    • What is the possible value of size?
  ‣ $\text{if ( size } \geq 1000+\text{BASE\_SIZE })$ ... {
    • 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);
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