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
  ‣ Attacker takes control of control-flow
    • E.g., return address overwrite through buffer overflow

• Data attacks
  • E.g., critical variable overwrite through buffer overflow

• Program-specific input handling attacks
  ‣ E.g., SQL injection attacks

• Attacks during interaction with system resources
  ‣ E.g., symbolic link attacks, TOCTTOU race conditions

• Several program vulnerabilities allow attackers to achieve above attacks
  ‣ See CWE (http://cwe.mitre.org/)
void myfunc()
{
    char string[16];
    printf("Enter a string\n");
    scanf("%s", string);
    printf("You entered: %s\n", string);
}

int main()
{
    myfunc();
}
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
ajhsoieurhgeskljdfghkljghsdjfhgsljdkjfhgskljrhgfdkj
You entered: ajhsoieurhgeskljdfghkljghsdjfhgsljdkjfhgskljrhgfdkj
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
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]

CSE543 - Introduction to Computer and Network Security
What Happened?

Stack

- main() parameters (argc, argv)
- return address
- saved frame pointer
- main() local vars
- myfunc() parameters (void)
- return address
- sghfjds
- saved frame pointer
- gjlkhrfd
- 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");
    (line)
    _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)
return address
pc + 8
saved frame pointer
my evil code
myfunc() local vars
more evil code
string[16]
A First Defense

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

• What if we made the stack non-executable?
  ‣ AMD NX-bit
  ‣ More general: \( W \oplus X \)
A First Defense

• What if we made the stack non-executable?
  ‣ AMD NX-bit
  ‣ More general: \( W \ (\text{xor}) \ X \)
• Attacker response
  ‣ return-to-libc
A First Defense

• What if we made the stack non-executable?
  ‣ AMD NX-bit
  ‣ More general: \( W \ (\text{xor}) \ X \)

• Attacker response
  ‣ return-to-libc

myfunc() parameters (void)

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

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

• Attacker response
  ‣ return-to-libc

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

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

 pc of libc call()

 saved frame pointer

 arguments for libc call

 myfunc() local vars

 string[16]
**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?
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?
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

<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></td>
</tr>
<tr>
<td>myfunc()</td>
<td>(void)</td>
<td></td>
</tr>
<tr>
<td>main() local vars</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>saved frame pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>myfunc() parameters (void)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>saved frame pointer</td>
<td></td>
<td>???</td>
</tr>
<tr>
<td>myfunc() local vars</td>
<td></td>
<td>???</td>
</tr>
<tr>
<td>string[16]</td>
<td></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>
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 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

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

```
Chunks1, 2, and 3 are joined by a doubly-linked list
ps sz fd bk chunk1 ps sz fd bk chunk2 ps sz fd bk chunk3

Chunk2 may be unlinked by rewriting 2 pointers
ps sz fd bk chunk1 ps sz fd bk chunk2 ps sz fd bk chunk3

Chunk2 is now unlinked
ps sz fd bk chunk1 ps sz fd bk chunk3
```

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

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

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

free(chunk2) -->
assert(chunk2->prev->next == chunk2)
assert(chunk2->next->prev == chunk2)
Heap Overflow Defenses

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

```c
free(chunk2) -->
assert(chunk2->prev->next == chunk2)
assert(chunk2->next->prev == chunk2)
```
Heap Overflow Defenses

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

• Sanity checks during heap management

  \texttt{free(chunk2)} \rightarrow

  \texttt{assert(chunk2->prev->next == chunk2)}

  \texttt{assert(chunk2->next->prev == chunk2)}
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->prev->next == chunk2)
assert(chunk2->next->prev == chunk2)
```
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->prev->next == chunk2)
assert(chunk2->next->prev == chunk2)
```
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->prev->next == chunk2)
  assert(chunk2->next->prev == chunk2)
  ```
  - Added to GNU libc 2.3.5
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
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
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
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
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
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
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
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);
}

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 < 1000 + 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
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
```
 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';
```
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';
```
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
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
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