F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 2 — 2018/19

[0] No proprietary software has been used in producing these slides

Outline

Lecture 7: Code Security: Buffer Overflow Attacks

- **Code Security** deals with writing code that is “secure” against attacks, i.e. that cannot be tricked in performing an unintended task.
- This is important across all application domains, e.g. web programming, server programming, embedded systems programming.
- It is particularly important in embedded systems programming, because you often don’t have OS protection against attacks.
- You will learn more about security in **F20CN: Computer Network Security**.
- Here we focus on the **security of low-level code** and in particular on **buffer overflow attacks**.
- **NB:** Buffer overflow attacks are some of the most commonly occuring security bugs

Dynamically Changing Attributes: `setuid`

Background: dynamically changing the ownership of programs.

- Sometimes we want to specify that a file can only be modified by a certain program.
- Thus, we want to control access on a per-program, rather than a per-user basis.
- We can achieve this by creating a new user, representing the role of a modifier for these files.
- Mark the program, as `setuid` to this user.
- This means, no matter who started the program, it will run under the user id of this new user.
- Example:

User	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	r	r
Alice	rx	x	–	–
Accounts program	rx	r	rw	w
Bob	rx	r	r	r

- **Beware:** `setuid` programs are a major security pitfall!

[0] Figure from “Security Engineering” by Ross Anderson, Chapters 4, 6

Example code for setuid

```
static uid_t euid, uid;
int main(int argc, char * argvp[]) {
  FILE *file;
  /* Store real and effective user IDs */
  uid = getuid();  euid = geteuid();
  /* Drop privileges  */
  seteuid(uid);
  /* Do something useful ...  */
  /* Raise privileges, in order to access the file */
  seteuid(euid);
  /* Open the file; NB: this is owned and readable only by a different user *
  file = fopen("/tmp/logfile", "a");
  /* Drop privileges again */
  seteuid(uid);
  /* Write to the file */
  if (file) {
    fprintf(file, "Someone used this program: UID=%d, EUID=%d\n", getuid(), g
  } else {
    fprintf(stderr, "Could not open file /tmp/logfile; aborting ...\n");
    return 1;
  }
  /* Close the file and return */
  fclose(file);  return 0; }
```


Testing this prgram

As normal user do the following:

```
# do everything in an open directory
> cd /tmp
# download the source code
> wget http://www.macs.hw.ac.uk/~hwloidl/Courses/F21CN/Labs/OSsec/setuid1.c
# compile the program
> gcc -o s1 setuid1.c
# change permissions so that everyone can execute it
> chmod a+x s1
# check the permissions
> ls -lad s1
-rwxrwxr-x 1 hwloidl hwloidl 10046 2011-11-11 22:06 s1
# generate an empty logfile
> touch /tmp/logfile
# change permissions to make it read/writeable only by the owner!
> chmod go-rwx /tmp/logfile
# check the permissions
> ls -lad /tmp/logfile
-rw------- 1 hwloidl hwloidl 0 2011-11-11 22:06 /tmp/logfile
```


As guest user do the following

```
> cd /tmp
# try to run the program
> ./s1
Could not open file /tmp/logfile; aborting ...
# this failed, because guest doesn't have permission to write to logfile
```

As normal user do the following

```
# set the setuid bit
> chmod +s s1
> ls -lad s1
-rwsrwsr-x 1 hwloidl hwloidl 10046 2011-11-11 22:06 s1
```

Now, as guest you can run the program:

```
> ./s1
# now this succeeds, although the user still cannot read the file
> cat /tmp/logfile
cat: /tmp/logfile: Permission denied
```

But the normal user can read the file, eg:

```
> cat /tmp/logfile
Someone used this program: UID=1701, EUID=1701
Someone used this program: UID=12386, EUID=12386
```


Buffer Overflow Attacks

- Often low-level programs use fixed-size arrays (buffers) to store data.
- When copying into such buffers, the program has to check that it doesn't exceed the size of the buffer.
- There are no automatic bounds checks in low-level languages such as C.
- If no check is performed, the program would just overwrite the following data block.
- If the data beyond the bound is chosen to be malign, executable machine code, an attacker can gain control of the system in this way.

Example 1: Rsyslog

The following vulnerability in the `rsyslog` program was reported in Linux Magazin 12/11:

```
[...]
int i;  /* general index for parsing */
uchar bufParseTAG[CONF_TAG_MAXSIZE];
uchar bufParseHOSTNAME[CONF_HOSTNAME_MAXSIZE];
[...]
while(lenMsg > 0 && *p2parse != ':' && *p2parse != ' ' &&
      i < CONF_TAG_MAXSIZE) {
 bufParseTAG[i++] = *p2parse++;
 --lenMsg;
}
if(lenMsg > 0 && *p2parse == ':') {
 ++p2parse;
 --lenMsg;
 bufParseTAG[i++] = ':';
}
[...]
bufParseTAG[i] = '\0';  /* terminate string */
```


Example 2:

The following vulnerability in the `rsyslog` program was reported in Linux Magazin 12/11:

```
[...]
int i;  /* general index for parsing */
uchar bufParseTAG[CONF_TAG_MAXSIZE];
uchar bufParseHOSTNAME[CONF_HOSTNAME_MAXSIZE];
[...]
while(lenMsg > 0 && *p2parse != ':' && *p2parse != ' ' &&
      i < CONF_TAG_MAXSIZE) {
 bufParseTAG[i++] = *p2parse++;
 --lenMsg;
}
if(lenMsg > 0 && *p2parse == ':') {
 ++p2parse;
 --lenMsg;
 bufParseTAG[i++] = ':';
}
[...]
bufParseTAG[i] = '\0';  /* terminate string */
```


Discussion

- The goal of this code is to read tags and store them in a buffer.
- The program reads from a memory location `p2parse` and writes into the buffer `bufParseTAG`.
- The fixed size of the buffer is `CONF_TAG_MAXSIZE`
- The while-loop iterates over the input text, and also checks whether the index `i` is still within bounds.
- **BUT:** after the while loop, 1 or 2 characters are added to the buffer as termination characters; this can cause a buffer overflow!
- The impact of the overflow is system-specific. It can lead to overwriting the variable `i` on the stack.

Smashing the Stack

- One common form of exploiting a buffer overflow is to manipulate the stack.
- This can happen through unchecked copy operations into a local function variable or argument.
- This is dangerous, because local variables are kept on the stack, together with the return address for the function.
- Therefore, a buffer-overflow can directly **modify the control-flow** in the program.

Example of Smashing the Stack

Assume, we call this function:

```
int function() {
 int a;
 char b[5];
 char c[4];
 ...
}
```

The stack-layout for this function is:

```
c
b
a
...
return address
```

A buffer overflow of b can overwrite the contents of a, or maybe even the return address, which would change the control flow of the program.
Stack Guard and other security programs re-order the variables on the stack, and add variables at the end to detect overwrites.

Difficulties in exploiting the vulnerability

- The attacker needs to locate the position of the return address, and write the address of its own, malign code there.
- Several techniques can be used to achieve this.
- In a return-to-libc attack, the attacker overwrites the return address with a call to a known libc library function (eg. `system`).
- After this, the return address to the malign code and data for the arguments to the libc function is placed.
- This will cause a call to the libc function, followed by executing the malign code itself.

A Worst Case Scenario

A particularly dangerous combination of weaknesses is the following:

- A setuid function, raising privileges temporarily,
- which contains a buffer overflow vulnerability,
- and an attacker that plants shellcode as malign code onto the stack.
- If successful, the shellcode will give the attacker access to a full shell with the privileges used in that part of the application.
- If these are root privileges, the attacker can do anything he wants!

Prevention Mechanisms

- Canary variables, eg. on the stack, can detect overflows.
- Re-ordering variables on the stack can help to reduce the impact of a buffer overflow.
- Compiler modifications can change the pointer semantics, eg. never store a pointer directly, but only a version that needs to be XORed to get to the real address.
- Some operating systems allow to mark address blocks as non-executable.
- Address randomisation (re-arranging data at random in the address space) is frequently in modern operating systems to make it more difficult to predict where to find a return address or similar, attackable control-flow data.

Listing 2: imap/nntpd.c

Another attack mentioned in Linux Magazin 12/11 is this one:

```
do {
    if ((c = strrchr(str, ',')))
     *c++ = '\0';
 else
     c = str;

 if (!(n % 10)) /* alloc some more */
     wild = xrealloc(wild, (n + 11) * sizeof(struct wildmat));

 if (*c == '!') wild[n].not = 1;  /* not */
 else if (*c == '@') wild[n].not = -1; /* absolute not (feeding) */
 else wild[n].not = 0;

 strcpy(p, wild[n].not ? c + 1 : c);
 wild[n++].pat = xstrdup(pattern);
} while (c != str);
```


Listing 2: imap/nntpd.c

Another attack mentioned in Linux Magazin 12/11 is this one:

```
do {
    if ((c = strrchr(str, ',')))
     *c++ = '\0';
 else
     c = str;

 if (!(n % 10)) /* alloc some more */
     wild = xrealloc(wild, (n + 11) * sizeof(struct wildmat));

 if (*c == '!') wild[n].not = 1;  /* not */
 else if (*c == '@') wild[n].not = -1; /* absolute not (feeding) */
 else wild[n].not = 0;

 strcpy(p, wild[n].not ? c + 1 : c);
 wild[n++].pat = xstrdup(pattern);
} while (c != str);
```


Discussion

- This example is part of an IMAP server for emails.
- This code segment handles wildcards to perform operations.
- Its weakness is that it uses `strcpy` to copy a block of characters, which copies an **unbounded** 0-terminated block of memory.
- Instead, the function `strncpy` should be used, which takes the size of the block to copy as additional argument.