Embedded Systems

Mini-CTF

Project report

Noam Cohen
Ron Haim Hodadedi

Year 2022 semester A

Software Engineering at the JCT institution

Instructor: Arie Haenel

Contents:

Problem & Solution:

In the last period, especially in the last years, there is a great growth of technology in the world. However, there is also an immense germination in the world of Security. The world of Information Security is one of the most important problems today. At the same time as the advancement of technology, there is a growing need to address privacy and security issues. Every day we hear about a new Cyber-Attack on big governments, companies, citizens, etc. Most governments set themselves every day one very important goal: The privacy and security of the citizens. As this need grows, also grows the need to prepare the bases, armies, big or important companies to scenarios of Cyber-Attacks, or even to initiate Cyber-Attacks.
For all of this, there are some big problems: Money, Time and Resources.
Not all governments, organizations or companies have a budget to prepare their citizens, employees or bases to scenarios of Cyber-Attacks, and that causes a lot of security vulnerabilities, which attract them a lot of hackers that will try to hack their systems and bases - or sometimes enemy countries that will try to get as much information as possible or shut down your systems and bases.
Also, many times there is no time and resources to prepare people to attack or protect systems.
All of this brings us to one simple and genius solution - CTF.

So, what is 'CTF'

CTF (Capture the Flag) is a kind of information security competition that challenges contestants to solve a variety of tasks ranging from a scavenger hunt on Wikipedia to basic programming exercises, to hacking your way into a server to steal data. In these challenges, the contestant is usually asked to find a specific piece of text that may be hidden on the server or behind a webpage. This goal is called the flag, hence the name.
Like many competitions, the skill level for CTFs varies between the events. Some are targeted towards professionals with experience operating on cyber security teams. These typically offer a large cash reward and can be held at a specific physical location. Other events target the high school and college student range, sometimes offering monetary support for education to those that place highly in the competition.

Common types of CTF:

There are three common types of CTFs: Jeopardy, Attack-Defense and mixed.

Jeopardy-style CTFs have a couple of tasks in a range of categories. For example: Web, Forensic, Crypto, Binary, etc. In this type of present competitors with a set of questions that reveal clues that guide them in solving complex tasks in a specific order. By revealing clues, contestants learn the right direction regarding techniques and methodologies that are needed going forward. Teams receive points for each solved task. The more difficult the task, the more points you can earn upon its successful completion. The next task in the chain can be opened only after some team solves the previous task. Then the game time is over and the sum of points shows you a CTF winner.

Attack-Defense: This style of competition is much closer to the backyard capture the flag game than the Jeopardy style. In these types of events, teams defend a host PC while still trying to attack opposing teams' target PCs. Each team starts off with an allotted time for patching and securing the PC, trying to discover as many vulnerabilities as possible before the opponent attacking teams can strike. The team with the most points win.

Mixed competitions may vary possible formats. It may be something like a wargame with special time for task-based elements.

CTF games often touch on many other aspects of information security: **cryptography**, **stego**, **binary analysis**, **reverse engineering**, **mobile security** and others. Good teams generally have strong skills and experience in all these issues.

CTF categories:
Challenges are typically divided into 5 categories for CTF. Common the types of challenges are:
Web: This type of challenge focuses on finding and exploiting the vulnerabilities in web applications. They may test the participants' knowledge on SQL Injection, XSS (Cross-Site Scripting), and many more.

- **Forensics:** Participants need to investigate some sort of data, like do a packet analysis on a .pcap file, memory dump analysis, and so on.

- **Cryptography:** Challenges will focus on decrypting encrypted strings from various types of cryptography such as Substitution crypto, Caesar cipher, etc.

- **Reverse Engineering:** Usually requires participants to explore a given binary file whether PE file, ELF file, APK, EXE or some types of other executable binary. Participants need to find the key by decompilation, disassemble using static or dynamic analysis, or other reverse engineering tools.
- **OSINT:** The OSINT idea is to see how much information is available to you and understand the underlying hint's hidden in the challenges it-self with the help of google and bit problem-solving skills. So more tools like sherlock, and no focus on domain enumeration, etc.

What is our part in this?

As part of this course, we offer to build an **Embedded Mini-CTF**, as part of it will examine the security level's components on a Tiva Launchpad development board.
Because we do not exactly have a system to check and explore, we will create some challenges on this board (Tiva launchpad), with each challenge containing a vulnerability(ies) that the rest of the students will need to find.
Information security Researchers, will be the course participants - and based on their degree of success, they will be rated and the top 3 researchers will be considered as winners.

The degree of success will be measured by:

- Time to find the vulnerabilities.
- The **Risk level** of this vulnerability.

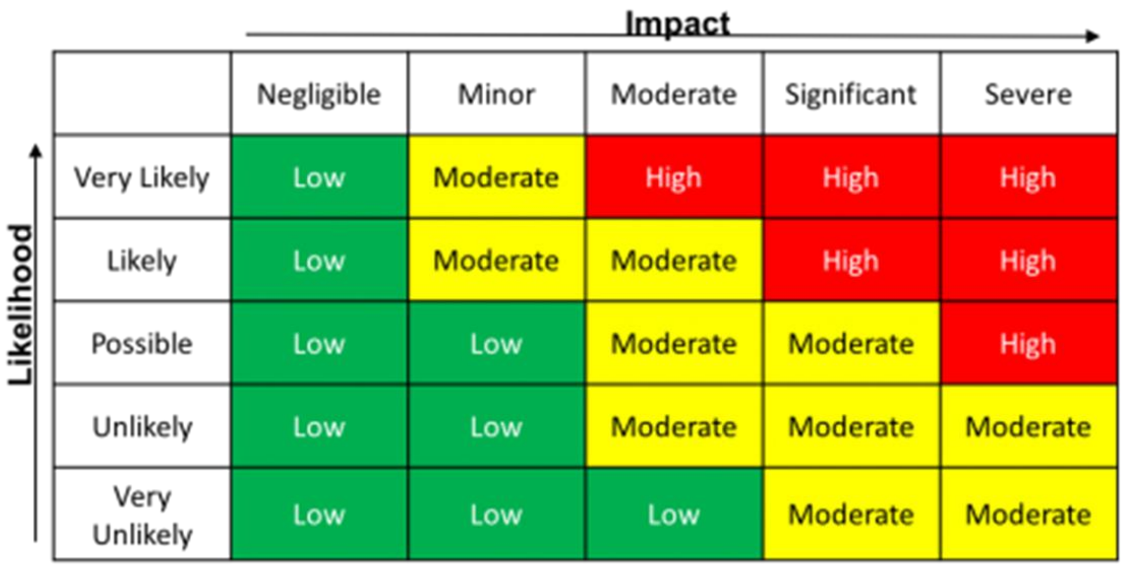
Risk level:

Each vulnerability will be rated with one of this risk ranks:

- High
- Medium
- Low

The risk level will be calculated according to the following parameters:

- **Attack potential**
  - The required knowledge level for the vulnerability exploit (Basic, Medium, Expert)
  - The required time for the vulnerability exploit (Minutes, Hours, Days)
  - Required equipment type for the vulnerability exploit (Standart, Special)
- **Damage potential**:
  The potential damage and the impact of exploiting the vulnerability (For example: Is this vulnerability can shut down the component or expose a secret information).


Impact
Likelihood

	Negligible	Minor	Moderate	Significant	Severe
Very Likely	Low	Moderate	High	High	High
Likely	Low	Moderate	Moderate	High	High
Possible	Low	Low	Moderate	Moderate	High
Unlikely	Low	Low	Moderate	Moderate	Moderate
Very Unlikely	Low	Low	Low	Moderate	Moderate

Project description:

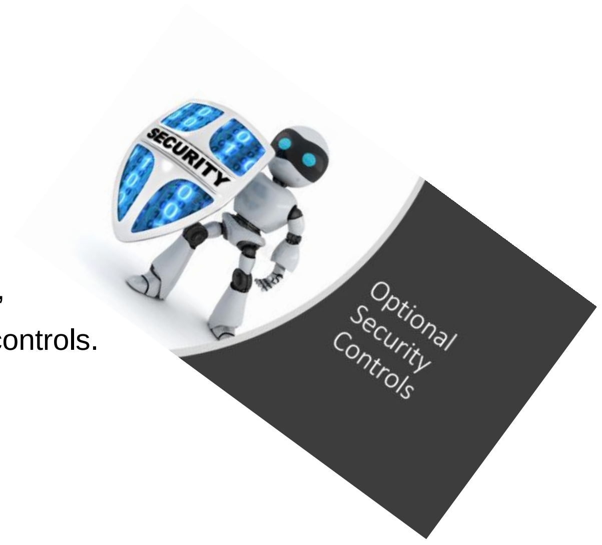
Subjects: ARM, Assembly, CTF, embedded development, exploitation

Creating some Security challenges on a Tiva Launchpad development board.

Before we will start to create our CTF we tried to find which vulnerability we can take advantage of on Tiva Launchpad.

The vulnerability that we choses to take (for now) is:

- Buffer overflow
- Format string attack.

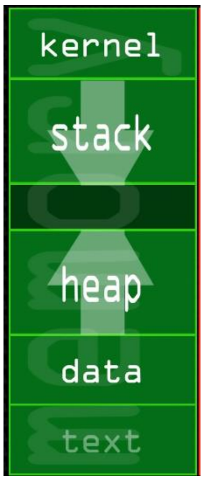


In this project proposal we will present those vulnerabilities, with a few examples of action ways, and optional security controls.

Note: Those vulnerabilities we chose, can be changed according to our schedule.

Buffer Overflow:

A buffer overflow exploit is a situation where we're using some, probably low-level C function or something to write a string or some other variable into a piece of memory that is only a certain length. But we're trying to write something in that's longer than that and it then overwrites the later memory addresses, and that can cause all kinds of problems. The first thing we should talk about, probably, is roughly what happens in memory with a program when it's run.
We have a big block of RAM. We don't know how big our RAM is because it can be varied, but we use something called Virtual Memory Address Translation to say that everything in here, this is 0. 0x00000000.
This is the bottom of the memory, as it were, and up here is 0xFFFFFFFF, this is the equivalent of "11111111" memory address all the way up to 32 or 64 bits.
Now, there are certain areas of this memory that are always allocated to certain things. So, we have the kernel, command line parameters that we can pass to our program and environment variables, and has something called the text, That's the actual code of our program.
The machine instructions that we've compiled get loaded in there.
We have the heap, it's where you allocate large things in your memory.
A big area of memory that you can allocate huge chunks on to do various things.



perhaps the most important bit, in some ways anyway, is the stack.
Now the stack holds the local variables for each of your functions and when you call a new function like, let's say you say "printf" and then some parameters that get put on the end of the stack.
So, the heap grows as you add memory, and the stack grows in this direction.
We have some program that's calling a function. A function is some area of code that does something and then returns to where it was before.
So, this is our calling function here. When the calling function wants to make use of something, it adds the parameters that it's passing onto the stack. it will be parameter A and parameter B, and they will be added into the stack in reverse order. And then the Assembler code for this function will make something called a "call" and that will jump to somewhere else in memory and work with these two parameters.

Let's look at some code and then we'll see how it works.

```
#include <stdio.h>
#include <string.h>

int main(int argc, int** argv)
{
	char buffer[500];
	strcpy(buffer, argv[1]);

	return 0;
}
```

It's a very simple C code that allocates some memory on the stack and then copies a string into it from the command line. we've got the main function for C that takes the number of parameters given and a pointer to those variables that you've got.
And they'll be held in the kernel area of our memory. We've allocated a buffer that's 500 characters long and then we call a function called "string copy" (strcpy) which will copy our command line parameter from argv into our buffer. Our function puts on a return address which is replacing the code we need to go back to once we've done strcpy. So that's how main knows where to go after it's finished. And then we put on a reference to the base pointer in our previous function. This is just going to be our EBP base pointer. This is our allocated space for our buffer, and it's 500 long. If we write into it something that's longer than 500, we're going to go straight past the buffer, and crucially over our return variable. then we point back to something we shouldn't be doing.

We can run our vulnerable code with "Hello".
And that will copy "Hello" into this buffer and then simply return, so nothing happens. It's the most boring program ever!

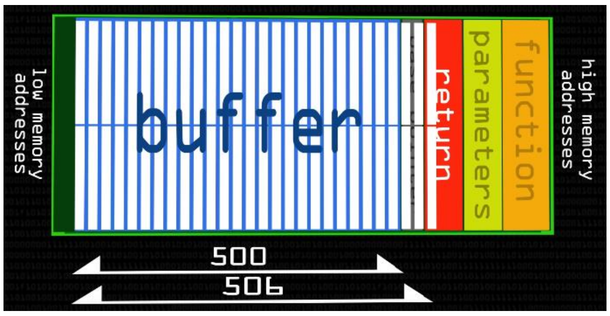
Q: Another program might do something like copy "Hello" in there and now it's in the buffer they can go off and process it?
A: Yes, maybe you've got a function that makes things all uppercase.
So, you copy "Hello" off and then you change this new copy to be all uppercase, and then you output it to the screen. And this doesn't have to be "main()", this could be any function.

Let look on assembly code

```
push    ebp
mov     ebp, esp
sub     esp, 380h
push    ebx
push    esi
push    edi
```

The line, sub esp, 380h, that's allocating the 500 for the buffer. That is, we're here and we go 500 in this direction and that's where our buffer goes. So, buffer's sitting to the left in this image but is lower in memory than the rest of our variables. We can run this program, and we can look at the registers and find out what's happened. So, we can say run "Hello" and it will start the program and say "Hello", and it's exited normally. Now, we can pass something in a little bit longer than "Hello". If we pass something that's over 500, then this buffer will go over base pointer and this return value and break the code.

Q: So that will just crash it?
A: It should just crash it.

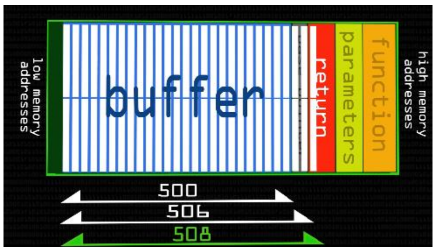


Now let's put in 506 times the "A" character", just a little bit more than 500 so it's going to cause somewhat of a problem but not a catastrophe. It will receive a segmentation fault, segmentation fault is what a CPU will send back to you when you're trying to access something in memory you shouldn't be doing.

Now that's not actually happened because we overwrote somewhere we shouldn't; what's happened is the return address was half overwritten with these 41's.

Q: So, it doesn't know what it is?
A: Yes, there is nothing in memory at 0xb7004141, and if there is, it doesn't belong to this process. It's not allowed, so it gets a segmentation fault.

So, let's change this to 508, we're going two bytes further along, which means we're now overwriting the entirety of our return address.



We're overwriting this "ret" with 41s. Now if there were some virus codes at 414141, that's a big problem. So that's where we're going with this.

So, we run this, and you can see the return address is now 0x414141.

Look on the registers and you can see that the construction pointer is now trying to point to 0x414141. So that means that it's read this return value and tried to return to that place in the code and run it, and of course it can't.

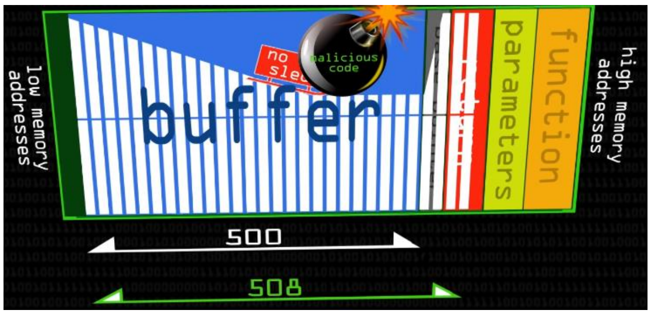
Now we can have a little bit more fun. We've broken our code, what can we do now?

We need to do is change this return value to somewhere where we've got some payload we're trying to give- we're trying to produce.

Now in fact this payload is just a simple, very short program in Assembler, that puts some variables on the stack and then executes a system call to tell it to run a shell to run a new command line. Crucially, ecd / x80 is throwing a system interrupt, which means that it's going to run the system call. We're going to put in our \x41's times by 508 - and then we're going to put in our shell code. So now we're doing all 41s and then a bunch of malicious code. Now that's too long; we've gone too far. And finally, the last thing we want to add in is our return address, which we'll customize in a moment.

To craft an exploit from this, what we need to do is remember the fact that strcpy is going to copy into our buffer. So, we want to overwrite the memory of return address with somewhere pointing to our malicious code. Now, we can't necessarily know for sure where our malicious code might be stored elsewhere on the disc, so we don't worry about that or memory. We want to put it in this buffer. So, we're going to put some malicious code in here and then we're going to have a return address that points back into it.

Memory moves around slightly. When you run programs, things change slightly, environment variables are added and removed, things move around.
So, we want to try and hedge our bets and get the rough area that this will go in. In the start of the buffer, we put in something called a No-Op sled. Or there's various other words for it. This is simply \x90. That is a machine instruction for "just move to the next one", So that's good.



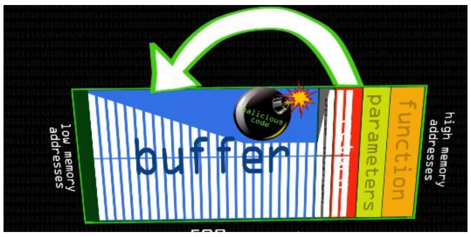
Anywhere we land in that No-Op is going to tick along to our malicious code.
So, we have a load of \x90s then we have our shell code.
That's our malicious payload that runs our shell.
And then we have the return address, right in the right place, that points back right smack in the middle of these \x90s. And what that means is, even if these move a bit, it'll still work.

Q: So, it's like having a slope almost, is it?
A: It's exactly like that, yes. Anywhere where we land in here is going to cause a real problem for the computer.
Q: So, we've got our bomb, pit of lava.
A:Yes, your No-Op sled takes you in and you get digested over 10,000 years or whatever it is.

So, we've got three things we need to do. We need to put in some \x90s, we need to put in our shell code, which I've already got, and we need to put in our return address.
Worry about the return address last.
If we go back to our code, we change the first \x41s that we were putting in, and we change to 90. We're putting in a load of No-Op operations.
Then we've got our shell code and then we've got what will eventually be our return address. And we'll put in 10 of those because it's just to have a little bit of padding between our shell code and our stack that's moving about.
So, if we write 508 bytes, it goes exactly where we want: over our return address.
But we've now got 43 bytes of shell code and we've got 40 bytes of return address.
So -40... -43 it is 425. We'll change this 508 to 425, and so now this exploit here that we're looking at is exactly what I hoped it would be here.

Some \x90 no operation sleds, the shell code and then we've got our return address, which is 10 times four bytes.
We run this and we've got a segmentation fault, which is exactly what we hoped we'd get because our return address hasn't been changed yet.
So now let's look at our memory and work out where our return address should go.

So, we can say "list the registers", let's say about 200 of them, at the stack point of -550.
So that's going to be right at the beginning of our buffer. And what we will see is a load of 90s in a row.
So, we just need to pick a memory address right in the middle of them, so let's pick this one, let's say 0xbffffaba.
Now, there's a nice quirk in this, which is that Intel CPUs are little endian, which means I must put it in backwards.
It's yet more things we must learn, but it's fine!
Theoretically when we run this, what will happen is string copy will do its thing: it'll copy a string in. And then, when it tries to return, it will load this return value and execute that instruction, which will be somewhere in this buffer.
Then it will read off and run our shell code, so we should get a shell.

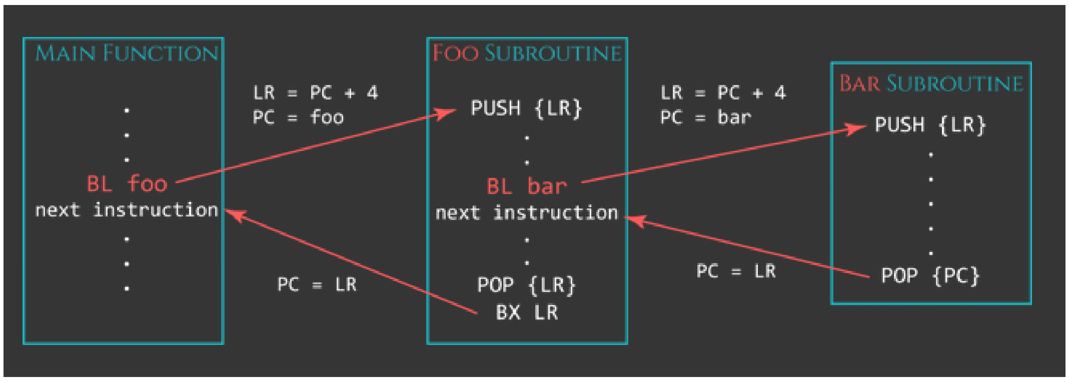


In ARM platforms what we describe doesn't working like that it is different from x86
let's see how it is work.

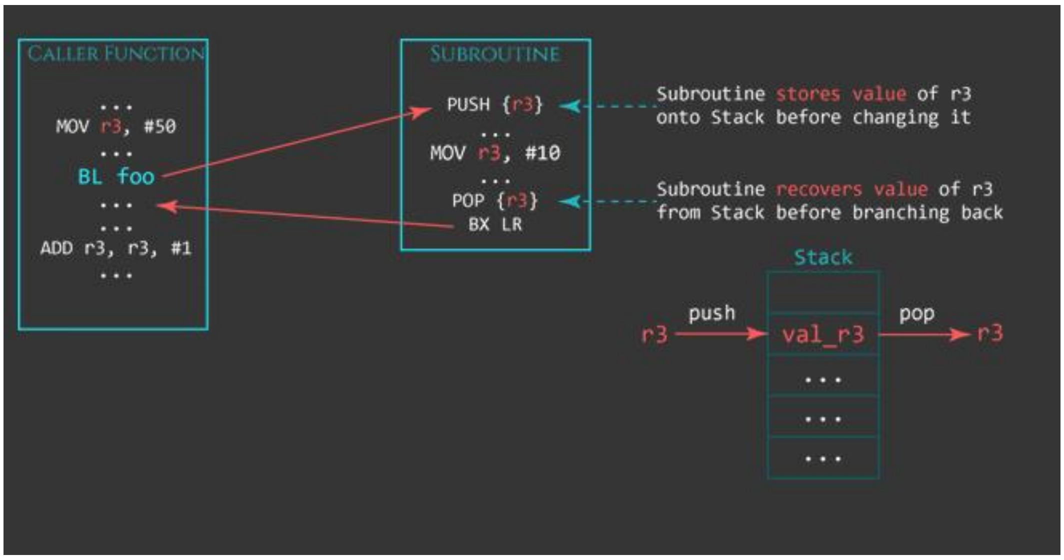
The following simple program will serve as an example:

```
#include <stdio.h>
#include <string.h>

void func1(char *s)
{
        char buffer[12];
        strcpy(buffer, s);
}

int main(int argc, char *argv[])
{
        if(argc > 1) {
                func1(argv[1]);
                printf("Everything is fine.\n");
        }
}
```

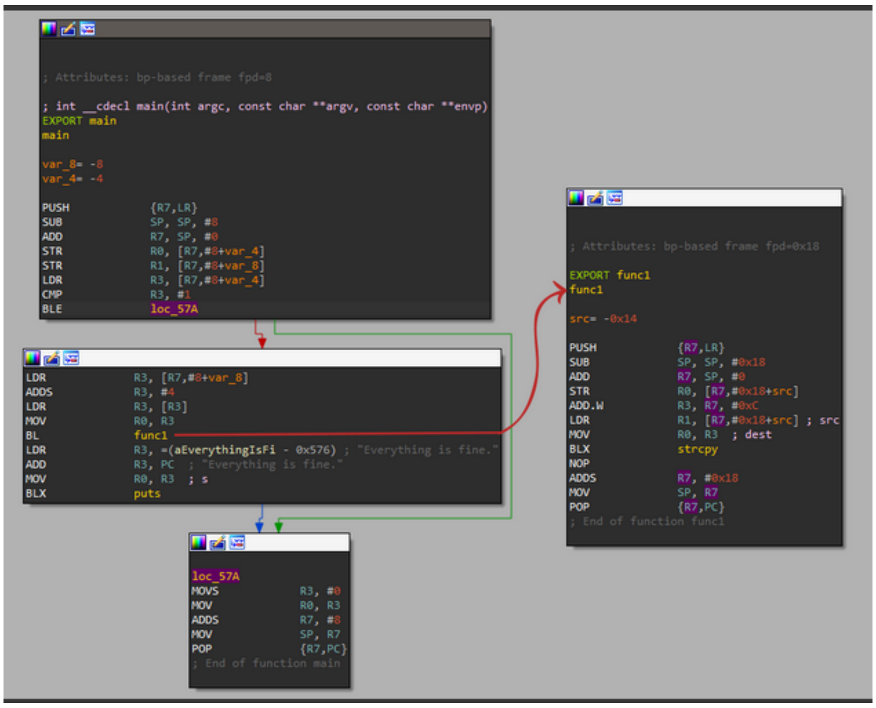
When a subroutine is being called, the return address is being preserved in the Link Register. This is done with a Branch with Link (BL) or Branch with Link and Exchange (BLX) instruction. But what if this subroutine calls another function? The Link Register would be overwritten, and the program would not find its way back to the previous function. The way this is handled is by preserving the return address on the stack with a PUSH instruction. The PUSH instruction stores the register it is given (in this case LR: push {LR}) to the top of the stack before overwriting the register LR with the new return address.



The same logic is used for registers that are being reused in a subroutine but need to keep track of the original value. In the next graphic you can see that the caller function uses register R3 as a counter, for example. The subroutine happens to require R3 for its own purposes and overwrites R3 with a different value and processes it in some way. To preserve the original value of R3, it gets pushed onto the stack at the beginning of the subroutine, changed and processed, and then restored to its original value by loading it from the top of the stack back to R3 with a POP instruction.


CALLER FUNCTION
MOV r3, #50
BL foo
ADD r3, r3, #1
SUBROUTINE
PUSH {r3}
MOV r3, #10
POP {r3}
BX LR
Subroutine stores value of r3 onto Stack before changing it
Subroutine recovers value of r3 from Stack before branching back
Stack
r3
push
val_r3
pop
r3

Let’s have a look at our stack overflow program. In func1, you can see that R7 and LR are being preserved on the stack because this function is about change the original value of R7 and call strcpy with a BLX instruction that overwrites LR with the new return address.



```
; Attributes: bp-based frame fpd=8

; int __cdecl main(int argc, const char **argv, const char **envp)
EXPORT main
main

var_8= -8
var_4= -4

PUSH            {R7,LR}
SUB             SP, SP, #8
ADD             R7, SP, #0
STR             R0, [R7,#8+var_4]
STR             R1, [R7,#8+var_8]
LDR             R3, [R7,#8+var_4]
CMP             R3, #1
BLE             loc_57A
```

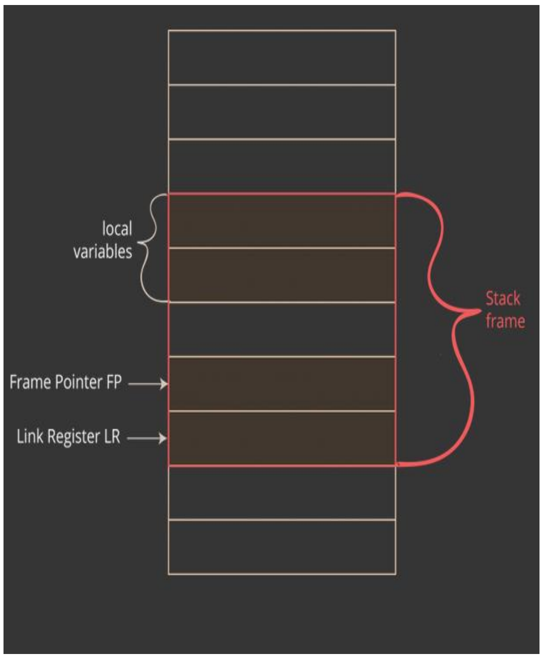
```
LDR             R3, [R7,#8+var_8]
ADDS            R3, #4
LDR             R3, [R3]
MOV             R0, R3
BL              func1
LDR             R3, =(aEverythingIsFi - 0x576) ; "Everything is fine."
ADD             R3, PC  ; "Everything is fine."
MOV             R0, R3  ; s
BLX             puts
```

```
loc_57A
MOVS            R3, #0
MOV             R0, R3
ADDS            R7, #8
MOV             SP, R7
POP             {R7,PC}
; End of function main
```

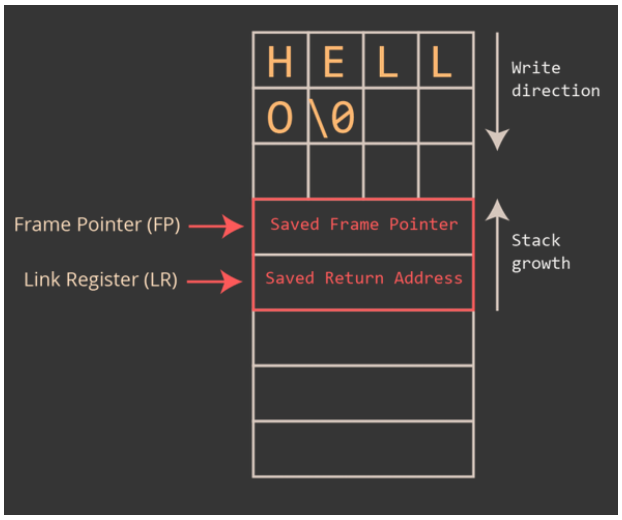
```
; Attributes: bp-based frame fpd=0x18

EXPORT func1
func1

src= -0x14

PUSH            {R7,LR}
SUB             SP, SP, #0x18
ADD             R7, SP, #0
STR             R0, [R7,#0x18+src]
ADD.W           R3, R7, #0xC
LDR             R1, [R7,#0x18+src] ; src
MOV             R0, R3  ; dest
BLX             strcpy
NOP
ADDS            R7, #0x18
MOV             SP, R7
POP             {R7,PC}
; End of function func1
```

Can you already guess how this logic gives us the ability to take control over the program-flow? Func1 first saves the return address stored in LR onto the stack, but at the end of the function it stores this value from the stack back to PC. What is PC used for again? PC is the register that holds the address of the next instruction to be executed. Convenient, isn’t it?

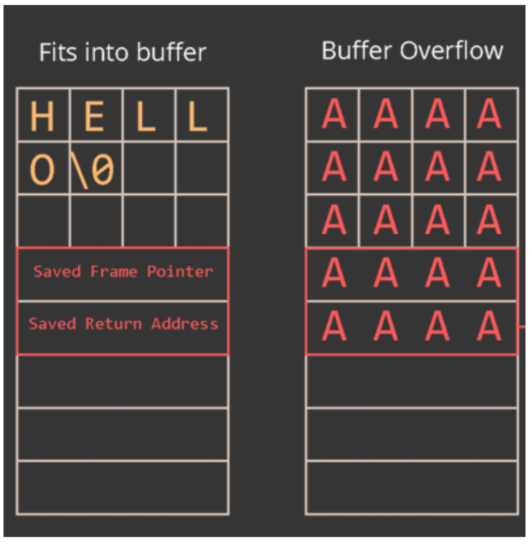
Let’s look at this from the stack perspective. Every function gets its own stack frame where it can store the return address and local variables, temporary values, etc. This function is also responsible for cleaning up after itself. This means that the first value it pushes onto the stack is the last value it pops back where it belongs.



How can we exploit this? In this example, a function stores the return address onto the stack and defines a local variable that is defined as a sequence of characters. The buffer is allocated by “moving” the Stack Pointer (SP) up by the number of bytes the buffer requires. The write direction is “down” (if you think of the stack growing “up”).



If the input string is “Hello”, it gets written into this allocated buffer and everything continues as expected. What if the input string is larger than the allocation can hold and there is no size check in place? You guessed it. If all elements of the buffer are filled, followed by characters that spill past the end of the buffer, the return address gets overwritten.



When the function ends and tries to restore program execution to its caller with the POP {PC} instruction, it will happily take whatever value at the position it expects the return address and store it in PC. In this case, program-flow is redirected to address 0x41414141 (AAAA). The program will try to execute the instruction at this address, but this address cannot be reached in memory space (as shown by executing *xinfo 0x41414141* in GDB/GEF) and the program aborts with a SIGSEGV, Segmentation fault.

Format string:

A software works according to a set of rules, that set by the developer. According to these rules, the process running in the device and do his job.
Exploitation of software vulnerabilities, is expressed in the rules set to get other result – the required result for the attacker, to cause the software act in his favor.
Format string attack is a tactic for exploitation of software vulnerability. Like Buffer overflow, the main goal is data override to get a control on the software, in our favor.
Format string usually happens because the programmer laziness.

What is a format function?
A format function is a special kind of ANSI C function, that takes a variable number of arguments, from which one is the so-called format string. While the function evaluates the format string, it accesses the extra parameters given to the function. It is a conversion function, which is used to represent primitive C data types in a human readable string representation. They are used in nearly any C program, to output information, print error messages, etc.

How does a format string vulnerability look like?
If an attacker can provide the format string to an ANSI C format function in part or as a whole, a format string vulnerability is present. By doing so, the behavior of the format function is changed, and the attacker may get control over the target application.

The format function family:

- **fprintf** – print to a file stream.
- **printf** – print to stdout stream.
- **sprintf –** print into a string.
- **snprintf** – print into a string, with length checking.
- **vfprintf** – print to a file stream from a va_arg structure.
- **vprintf** – print to stdout from a va_arg structure.
- **vsprintf** – print to a string from va_arg structure.
- **vsnprintf** – print to a string with length checking from a va_arg structure.
- **setproctitle** – set argv[]
- **syslog –** output to the syslog facility.

What exactly is a format string?

A format string is an ASCII string that contains text and format parameters.

For example:

```
#include <stdio.h>


void main()
{
	printf("The magic number is: %d\n", 1911);
}
```

In this example, the text to be printed is "The magic number is:" followed by a format parameter '%d', that is replaced with the parameter (1911) in the output. The output looks like: The magic number is: 1911.

Examples of format parameters:

- %d – print a decimal value.
- %u – print an unsigned decimal value.
- %x – print a hexadecimal value.
- %s – print a string by reference.
- %n – print the number of bytes written so far.

The '\' characters is used to escape special characters. It is replaced by the C compiler at compile-time, replacing the escape sequence by the appropriate character in the binary. The format functions do not recognize those special sequences. In fact, they do not have anything to do with the format functions at all, but are sometimes mixed up, as if they are evaluated by them.

For example:

```
#include <stdio.h>

void main()
{
	printf("The magic number is: \x25d\n", 23);
}
```

The code above works, because '\x25' is replaced at compile time with '%', since 0x25 (37) is the ASCII value for he precentpresents character.

The stack and its role at format strings:

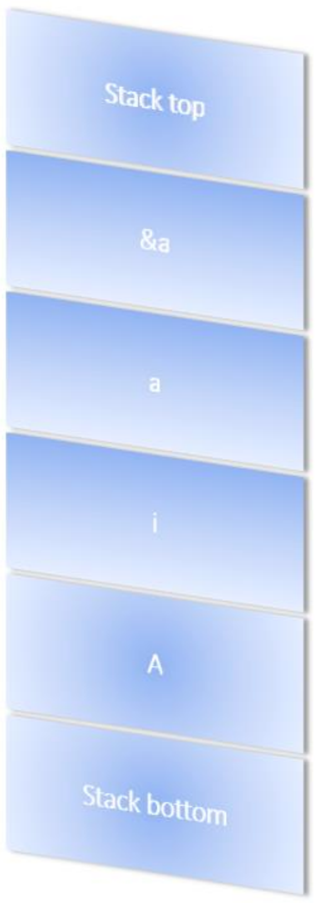
The behavior of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack.

Let's look about this code:

```
#include <stdio.h>


void main()
{
	printf("Number %d has no address, number %d has: %08x\n", i, a, &a);
}
```

From within the printf function the stack looks like:

- A – address of the format string.
- i – value of the variable i.
- a – value of the variable a.
- &a – address of the variable a.



The format function now parses the format string 'A', by reading a character a time. If it is not '%', the character is copied to the output.

In case it is, the character behind the '%' specifies the type of parameter that should be evaluated. The string "%%" has a special meaning, it is used to print the escape character '%' itself. Every other parameter relates to data, which is located on the stack.

Format string vulnerabilities:

One of the famous format string vulnerabilities is a **Channeling problem**.

this type of vulnerability can appear if two different types of information channels are merged into one, and special escape characters or sequences are used to distinguish which channel is currently active. Most of the times, one channel is a data channel, which is not parsed actively but just copied, while the other channel is a controlling channel.

While this is not a bad thing in itself, it can quickly become a horrible problem – if the attacker is able to supply input that is used in one channel. Channeling problems are no security holes itself, but they make bugs exploitation.

Through supplying the format string we are able to control the behaviour of the format function. We now have to examine what exactly we are able to control, and how to use this control to extend this partial control over the process to full control of the execution flow.

A simple attack using format string vulnerabilities is to make the process crash. This can be useful for some things. For example, in some network attacks it is useful to have a service not responding.

In another matter, let's look about this example:

```
#include <stdio.h>


void main()
{
	printf("%s%s%s%s%s%s%s%s%s%s%s%s");
}
```

Because '%s' displays memory from an address that is supplied on the stack, where a lot of other data is stored too our chances are high to read from an illegal address, which is not mapped. Also most format function implementations offer the '%n' parameter, which can be used to write to the addresses on the stack. If that is done a few times, it should reliably produce a crash.

If we can see the output string (the reply of the format function), we can get useful data from it, since it is the output of the behavior we control, and we can use these results to gain an overview of what our format string does and how the process layout looks like. This can be useful for various things, such as finding the correct offsets for the real exploitation or reconstructing the stack frames of the target process.

Viewing the stack:

We can show some parts of the stack memory by using format string like this:

```
#include <stdio.h>


void main()
{
	printf("%08x.%08x.%08x.%08x.%08x\n");
}
```

This works because we instruct the printf function to retrieve five parameters from the stack and display them as 8 digit padded hex numbers. Possible output will be something like this:

```
40012980.080628c4.bffff7a4.00000005.08059c04
```

It is also possible to peek at memory locations different from the stack memory. To do this we have to get the format function to display memory from an address we can supply. This poses two problems to us: First, we have to find a format parameter which uses an address (by reference) as stack parameter and displays memory from there, and we have to supply that address. We are lucky in the first case, since the '%s' parameter just does that, it displays memory — usually an ASCII string — from a stack supplied address. So the remaining problem is, how to get that address on the stack, into the right place.
Our format string is usually located on the stack itself, so we already have near to full control over the space, where the format string lies.
The format function internally maintains a pointer to the stack location of the current format parameter. If we would be able to get this pointer pointing into a memory space we can control, we can supply an address to the '%s' parameter. To modify the stack pointer we can simply use dummy parameters that will 'dig' up the stack by printing junk:

```
#include <stdio.h>
void main()
{
	printf("AAA0AAA1_%08x.%08x.%08x.%08x.%08x");
}
```

The '%08x' parameters increase the internal stack pointer of the format function towards the top of the stack. After more or less of this increasing parameter the stack pointer points into our memory: the format string itself. The format function always maintains the lowest stack frame, so if our buffer lies on the stack at all, it lies above the current stack pointer for sure. If we choose the number of '%08x' parameters correctly, we could just display memory from an arbitrary address, by appending '%s' to our string. In our case the address is illegal and would be 'AAA0'. Let's replace it with a real one.

```
address = 0x08480110
address (encoded as 32 bit le string): "\x10\x01\x48\x08"
#include <stdio.h>
void main()
{
	printf("\x10\x01\x48\x08_%08x.%08x.%08x.%08x.%08x|%s|");
}
```

Will dump memory from 0x08480110 until a NULL byte is reached. By increasing the memory address dynamically we can map out the entire process space. It is also helpful to find the cause of unsuccessful exploitation attempts.
If we cannot reach the exact format string boundary by using 4-Byte pops ('%08x'), we have to pad the format string, by prepending one, two or three junk characters.
For summary, we have mentioned just a little of the possible action ways in format string, and we will try to implement part of those ways (or another ways) in Embedded system.

Bonus options:
It's important to note, that these vulnerabilities we will use for our challenges, are an initial line and if necessary we will improve this to some other types of vulnerabilities, or we will append more challenges that will reflect our knowledge and techniques of using Embedded systems. In addition, we planning to add some functionality to our challenges, like a sound or a small LCD screen, a matrix to be used as a keyboard.

Side channel attack:

Q: So, what is side channel attack?

A: A side-channel attack is a security exploit that aims to gather information from or influence the program execution of a system by measuring or exploiting indirect effects of the system or its hardware -- rather than targeting the program or its code directly. Most commonly, these attacks aim to exfiltrate sensitive information, including cryptographic keys, by measuring coincidental hardware emissions.

Attackers can also go after high-value targets, such as secure processors, Trusted Platform Module (TPM) chips, and cryptographic keys. Even having only partial information can assist a traditional attack vector, such as a brute-force attack, to have a greater chance of success. Side-channel attacks can be tricky to defend against. They are difficult to detect in action, often do not leave any trace, and may not alter a system while it's running. Side-channel attacks can even prove effective against air-gapped systems that have been physically segregated from other computers or networks. Additionally, they may also be used against virtual machines (VMs) and in cloud computing environments where an attacker and target share the same physical hardware.

Let's give little Illustration, Let's imagine you're trying to determine where a person has driven their car. A typical attack channel would be to follow the car or use a Global Positioning System (GPS) tracker. A side-channel attack, on the other hand, would use measurements about the car to try and determine how it's used. For example, measuring changes in the amount of gas in the tank, car's weight, the heat of the engine or passenger compartment, tire wear, paint scratches and the like may reveal information about the use of the car, places or distances it has traveled, or what is stored in the trunk -- all done without directly affecting the car or alerting its owner that they are under investigation.

Types of side-channel attacks

Acoustic: The attacker measures the sounds produced by a device. Proof-of-concept (POC) attacks have been performed that can reconstruct a user's keystrokes from an audio recording of the user typing. Hackers can obtain some information by listening to the sounds

Electromagnetic :The earliest side-channel attacks were electromagnetic. The attacker measures the electromagnetic radiation, or radio waves, given off by a target device to reconstruct the internal signals of that device. Attackers focus modern side-channel attacks on measuring the cryptographic operations of a system to try and derive secret keys.

Optical: An attacker uses visual cues to gain information about a system. Although rarely used against computers, some POC attacks have been performed where audio can be systems use data caching and pre-fetching to improve performance. An attacker can abuse these systems to access information that should be blocked.

Timing: A bad actor uses the length of time an operation takes to gain information. The total time can provide data about the state of a system or the type of process it is running. For example , the attacker can compare the length of time of a known system to the victim system to make accurate predictions.

Power: A hacker can measure or influence the power consumption of a device or subsystem. By monitoring the amount and timing of power used by a system or one of its subcomponents, an attacker can infer the activity of that system. Some attacks may cut or lower power to cause a system to behave in a way beneficial to the attacker, similar to Plundervolt attacks.

Hardware weaknesses: Hackers can use physical characteristics of a system to induce a behavior, cause a fault or exploit data remanence, which is data that persists after deletion. Row hammering attacks happen when an attacker causes a change in a restricted area of memory by quickly flipping, or hammering, another area of memory located close by on the physical random-access memory (RAM) chip. Error correction code (ECC) memory can help prevent this attack. In a cold boot attack, the attacker quickly lowers the temperature of RAM, causing some of the information to be retained after power is removed so the attacker can read it back.

So how we can prevent a side-channel attack?

we can implement a few best practice mitigations that may help protect against side-channel attacks. These attacks usually require specific detailed knowledge of a system to execute; therefore, we should keep details related to implementation and vendors as a trade secret.

- Address space layout randomization (ASLR) can prevent some memory or cache-based attacks.
- Using business-grade equipment can also help to prevent systems from being exploited.
- Physical access to systems should be restricted as well.
- We can also keep sensitive systems in shielded Faraday cages, and power conditioning equipment can shield against power attacks.
- Increasing the amount of noise in a system will make it more difficult for an attacker to gain useful information

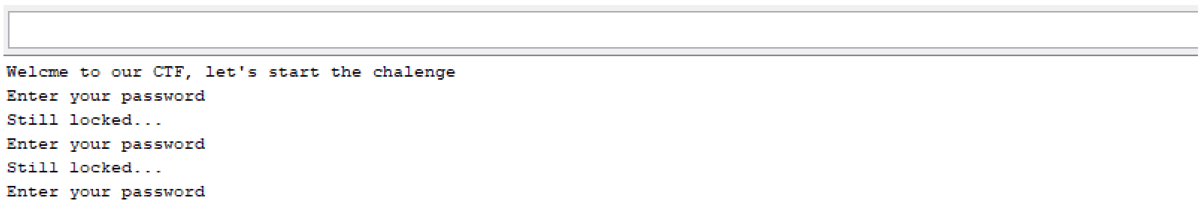
Before we start

After we researched the vulnerabilities, we start to research and think about how to realize them on our CTF challenge. We start by learning how to use different tools of the Arduino kit like LCD screen or matrix keyboard. To save time, we started to work with using Arduino processor because we know to work with him, and after that we will covert our project to Tiva processor.

Buffer overflow

Our first CTF challenge, Let's look about the C code[1]:
This level on this challenge will show a simple buffer overflow example that can be used to "unlock" something without knowing the password, by overwriting a lock flag.

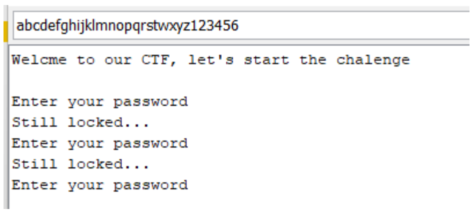
In the first line, we declare a local buffer of length 32 that will be used to be filled with user input for password. We also set up a local array of flags used to unlock various things in the codebase. These local variables are stored on the stack. Let's run the program an see what happening.

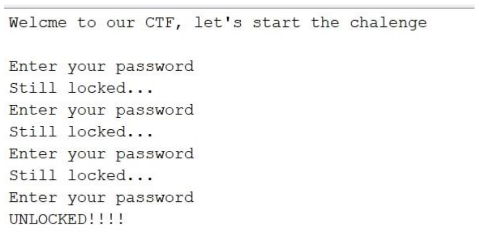

```
Welcme to our CTF, let's start the chalenge
Enter your password
Still locked...
Enter your password
Still locked...
Enter your password
```

Let look again on our code

```
char buff[BUFFER_LENGTH] = {0};
uint8_t unlocked_flags[64] = {0};
```

Memory allotment of the unloked_flags happens Immediately after we allotment Memory for char buffer. This means that if we can write beyond the 16 elements of the buffer, we can overwrite elements in the unlocked flags buffer. We can input a string with 18 characters, Let's try it.


```
abcdefghijklmnopqrstwxyz123456

Welcme to our CTF, let's start the chalenge

Enter your password
Still locked...
Enter your password
Still locked...
Enter your password
```


```
Welcme to our CTF, let's start the chalenge

Enter your password
Still locked...
Enter your password
Still locked...
Enter your password
Still locked...
Enter your password
UNLOCKED!!!!
```

Boom!!! We are got it!!!

[1] The C code in the appendices

Side channel attack:

These vulnerabilities we actualize on Arduino, we tried to convert the challenge for Tiva board, but we don't success. In this part of the project, we use at some of different tools from the Arduino kit like speaker for the sound, matrix keyboard for the keyboard and microphone for listening to the sound of any keyboard press[2].

The realization is having two parts, in the first level, the user insert is the password from the keyboard any bottom sounding a different voice when pressing on him. After that, the attacker listens to these sounds and then, he tries to restore the password from this voice.

First, we have a first Arduino board. This board connect to the matrix keyboard and each button sound a different frequency. 4 frequencies are the real password. In the second Arduino board we have the listener. The listener listen every time to 4 frequencies and compare this with the password. If we played the correct frequencies, we have got the password!

How it's happen? Let's look:

```
void makeButtonSound(char button)
{
  if (button == '1') { return makeSound(1000); }
  if (button == '2') { return makeSound(500); }
  if (button == '3') { return makeSound(600); }
  if (button == '4') { return makeSound(900); }
  if (button == '5') { return makeSound(1100); }
  if (button == '6') { return makeSound(700); }
  if (button == '7') { return makeSound(800); }
  if (button == '8') { return makeSound(1200); }
  if (button == '9') { return makeSound(1400); }
  if (button == '0') { return makeSound(1300); }
  //  if (button == 'A') { return makeSound(); }
  //  if (button == 'B') { return makeSound(); }
  if (button == 'C') { return makeSound(29); }
  if (button == 'D') {
    for (int i=20; i<30; i++){
      makeSound(i);
      delay(80);
    }
  }
}
```

We can see that every button sound a different frequency (5 times).

[2] The picture of the electric circuit is in the appendices

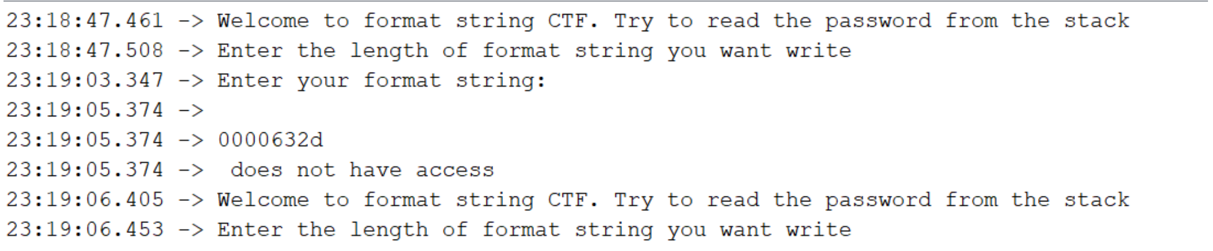
Format string:

Our last CTF challenge. In this CTF we need to implement a type of stack popping – try to read the password from the stack.

We can see the code in the attached.

In the first level we need to enter the length of the format string we want write (for the input process in serial). After that we write in our format string to try pop the string from the stack.

For example:



```
23:18:47.461 -> Welcome to format string CTF. Try to read the password from the stack
23:18:47.508 -> Enter the length of format string you want write
23:19:03.347 -> Enter your format string:
23:19:05.374 -> 
23:19:05.374 -> 0000632d
23:19:05.374 ->  does not have access
23:19:06.405 -> Welcome to format string CTF. Try to read the password from the stack
23:19:06.453 -> Enter the length of format string you want write
```

Here We try just tried kind of format string (the solution you need to find 😊) like %08x (just example…)

After we find the right format, the program will print:

`Fundmentals_Of_Software_Security does not have access`

(This is an example the password) And we got the password!

Bibliography:

- Exploiting format string vulnerabilities, scut / team teso, September 1, 2001: https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf
- INTRODUCTION TO STACK OVERFLOWS ON ARM3: https://azeria-labs.com/stack-overflow-arm32/
- Format string exploitation, Raziel Bakar: https://www.digitalwhisper.co.il/files/Zines/0x48/DW72-4-FormatString.pdf
- Format string exploitation – tutorial, Saif El-Sherei: https://www.exploit-db.com/docs/english/28476-linux-format-string-exploitation.pdf
- What is CTF and how to get started! Atan: https://dev.to/atan/what-is-ctf-and-how-to-get-started-3f04
- Running a buffer overflow attack – Computerphile: https://www.youtube.com/watch?v=1S0aBV-Waeo

Appendices

Buffer overflow challenge

```
#define BUFFER_LENGTH   32

void setup() {
  Serial.begin(115200);
  delay(1000);
   /*print welcome message*/
  Serial.println("Welcme to our CTF, let's start the chalenge\n");
  delay(100);
}
void loop() {
  delay(5000);
  char buff[BUFFER_LENGTH] = {0};
  uint8_t unlocked_flags[64] = {0};
  int start = millis();
  int index = 0;
  Serial.println("Enter your password");
  while((millis() - start) < 5000)
  {
    if(Serial.available())
    {
      buff[index++] = Serial.read();
    }
  }
  if(strcmp("super_secret", buff) == 0){
    Serial.println("Password correct!");
    unlocked_flags[0] = 1;
  }
  if(unlocked_flags[0]){
    Serial.println("UNLOCKED!!!!");
    delay(500);
    Serial.println("Let's keep on to next 1eve1");
    delay(500);
    Serial.print("Your key is: ");
    Serial.println((uint32_t) unlocked_flags);
    delay(5000);
    Serial.println("Save him, you will need him on the next level");
    delay(500000);
  }
  else  {
    Serial.println("Still locked...");
  }

}
```

Code of insert the password:

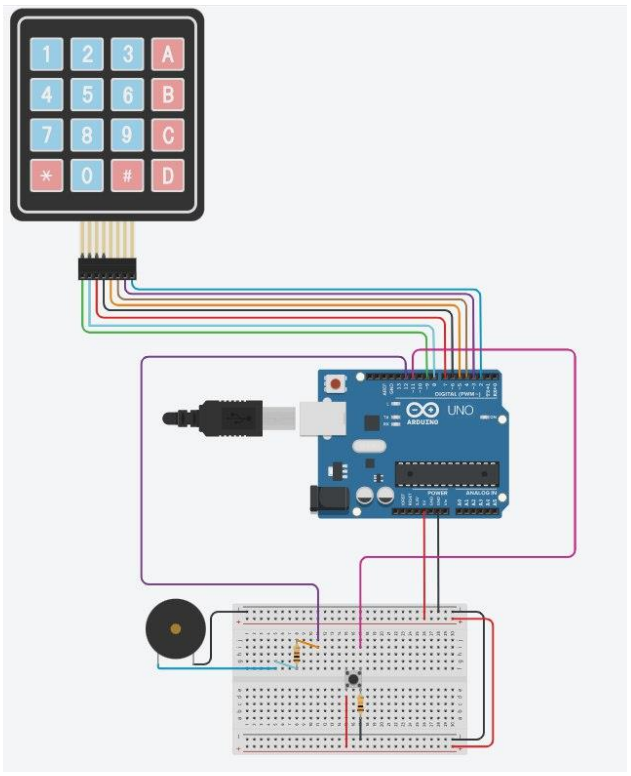
```
 Include the Keypad library
#include <Keypad.h>


const int buzzerPin = 12; // declaring the PWM pin</p><p>void setup() {

 Constants for row and column sizes
const byte ROWS = 4;
const byte COLS = 4;

 Array to represent keys on keypad
char hexaKeys[ROWS][COLS] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};
/*
  {'s4', 's8', 's12', 's16'},
  {'s3', 's7', 's11', 's15'},
  {'s2', 's6', 's10', 's14'},
  {'s1', 's5', 's9', 's13'}
*/


 Connections to Arduino
byte rowPins[ROWS] = {9, 8, 7, 6};
byte colPins[COLS] = {5, 4, 3, 2};

 Create keypad object
Keypad customKeypad = Keypad(makeKeymap(hexaKeys), rowPins, colPins, ROWS, COLS);

void setup() {
   Setup serial monitor
  Serial.begin(9600);
}

void loop() {
   Get key value if pressed
  char customKey = customKeypad.getKey();

  if (customKey) {

     Print key value to serial monitor
    Serial.println(customKey);

    makeButtonSound(customKey);
  }
}

 400, 300, 2000,

void makeButtonSound(char button) {
  if (button == '1') { return makeSound(1000); }
  if (button == '2') { return makeSound(500); }
  if (button == '3') { return makeSound(600); }
  if (button == '4') { return makeSound(900); }
  if (button == '5') { return makeSound(1100); }
  if (button == '6') { return makeSound(700); }
  if (button == '7') { return makeSound(800); }
  if (button == '8') { return makeSound(1200); }
  if (button == '9') { return makeSound(1400); }
  if (button == '0') { return makeSound(1300); }
  if (button == 'A') { return makeSound(); }
  if (button == 'B') { return makeSound(); }
  if (button == 'C') { return makeSound(29); }
  if (button == 'D') {
    for (int i=20; i<30; i++){
      makeSound(i);
      delay(80);
    }
  }
  if (button == '*') { return makeSound(); }
  if (button == '#') { return makeSound(); }
}



void makeSound(int indexOfBeep) {
Serial.begin(8600);
pinMode(buzzerPin, OUTPUT); //adding pin to Output mode</p><p>}</p><p>void loop() {
  tone(buzzerPin, 50);
  tone(buzzerPin, indexOfBeep);
  delay(500);
  delay(750);
  noTone(buzzerPin);
  delay(100);

  delay(250);

  tone(buzzerPin, 10);
  delay(50);
  noTone(buzzerPin);
  delay(100);
}
```

The code of attacker:

```
#include "arduinoFFT.h"

#define SAMPLES 128
#define SAMPLING_FREQUENCY 2048

arduinoFFT FFT = arduinoFFT()

unsigned int samplingPeriod
unsigned long microSeconds

double vReal[SAMPLES]
double vImg[SAMPLES]

// constants won't change. They're used here to set pin numbers:
const int buttonPin = 11
// the number of the pushbutton pin
const int ledPin = 13
// the number of the LED pin

// default of the password will be - 2...
int buttonPassword[4] = {-2, -2, -2, -2}

double button0Frequency = 1300
double button1Frequency = 1000
double button2Frequency = 500
double button3Frequency = 600
double button4Frequency = 900
double button5Frequency = 1100
double button6Frequency = 700
double button7Frequency = 800
double button8Frequency = 1200
double button9Frequency = 1400

int mapNoiseToButton(double noise, double treshold=30)
{

    if ((noise < (button0Frequency + treshold)) & & (noise > (button0Frequency
- treshold))) {return 0

                }
    if ((noise < (button1Frequency + treshold)) & & (noise > (button1Frequency
- treshold))) {return 1

                }
    if ((noise < (button2Frequency + treshold)) & & (noise > (button2Frequency
- treshold))) {return 2
```

```
                }
    if ((noise < (button3Frequency + treshold)) & & (noise > (button3Frequency
- treshold))) {return 3

                }
    if ((noise < (button4Frequency + treshold)) & & (noise > (button4Frequency
- treshold))) {return 4

                }
    if ((noise < (button5Frequency + treshold)) & & (noise > (button5Frequency
- treshold))) {return 5

                }
    if ((noise < (button6Frequency + treshold)) & & (noise > (button6Frequency
- treshold))) {return 6

                }
    if ((noise < (button7Frequency + treshold)) & & (noise > (button7Frequency
- treshold))) {return 7

                }
    if ((noise < (button8Frequency + treshold)) & & (noise > (button8Frequency
- treshold))) {return 8

                }
    if ((noise < (button9Frequency + treshold)) & & (noise > (button9Frequency
- treshold))) {return 9

                }

    return -1
}


// variables will change:
int currButtonState, prevButtonState = LOW
// variable for reading the pushbutton status
int ledState = LOW

void setup()
{
    // initialize the LED pin as an output:
    pinMode(ledPin, OUTPUT)
    // initialize the pushbutton pin as an input:
    pinMode(buttonPin, INPUT)


    Serial.begin(115200)
    // samplingPeriod = round(1000000 * (1.0/SAMPLING_FREQUENCY))
    samplingPeriod = round(489)
}

void loop()
{

    int indexOfPassword = 0
    for(int i=0
        i < 5
        i++)
    {

        // read the state of the pushbutton value:
        currButtonState = digitalRead(buttonPin)
        if (prevButtonState == LOW & & currButtonState == HIGH)
        {
            ledState = !ledState
            // turn LED on:
            digitalWrite(ledPin, ledState)
            delay(200)



            double noiseFrequency = 0
            for(int j=0
                j < 2
                j++)
            {
                noiseFrequency += listenToFrequency()
                Serial.println("the 1//2 frequency is:")
                Serial.println(noiseFrequency)
            }
            Serial.println("the total frequency is:")
            double finalNoiseFrequency = noiseFrequency/2
            Serial.println(finalNoiseFrequency)
            int currentButtonOfNoise = mapNoiseToButton(finalNoiseFrequency)
            if (currentButtonOfNoise != -1)
            {
                buttonPassword[indexOfPassword] = currentButtonOfNoise
                indexOfPassword += 1
                Serial.println(
                    "got a button!!! waiting a 2 second to the next one...\n")
                delay(1500)
                Serial.println("starting the next one")
                delay(500)
            }
            if (indexOfPassword >= 4)
      { // end of the length of the password - find all the buttons...
```

```
                break
            }
          }
          else
          {
              i -= 1
          }
    }
    Serial.println("progress done!!!\n\n")
    for(int i=0
        i < 4
        i++)
    {
        Serial.println(buttonPassword[i])
    }

    for(int i=1
        i < 5
        i++)
    {
        if (buttonPassword[i - 1] != i)
        {
            Serial.println("Wrong password.")
            break
        }

        else if (i == 4)
        {
            Serial.println("Correct password! the password is 1, 2, 3, 4")
        }
    }

    while(1)
}

double listenToFrequency()
{
    for(int i=0
        i < SAMPLES
        i++)
    {
        microSeconds = micros()

        vReal[i] = analogRead(0)
        vImg[i] = 0


        while(micros() < (microSeconds + samplingPeriod))
        {
            // do nothing
        }
    }

    FFT.Windowing(vReal, SAMPLES, FFT_WIN_TYP_HAMMING, FFT_FORWARD)
    FFT.Compute(vReal, vImg, SAMPLES, FFT_FORWARD)
    FFT.ComplexToMagnitude(vReal, vImg, SAMPLES)


    double peak = FFT.MajorPeak(vReal, SAMPLES, SAMPLING_FREQUENCY)
    Serial.println(peak)

    // while (1)
    Serial.println("Done.\n")
    delay(1000)
    Serial.println("starting more one...")

    if (peak < 485 | | peak > 1425)
    {
        Serial.println("below 485, or above 1425...")
        Serial.println(peak)
        Serial.println("\n")
        return listenToFrequency()
    }
    return peak
}
```



Format string attack code:

```
#define LENGTH 100

char buff[LENGTH];
char format[LENGTH];

void PrintWellcomeMessage();
void inputFormat();
void PrintResult(char* format);

char pass[] = "Fundmentals_Of_Software_Security";

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    PrintWellcomeMessage();
    delay(1000);

    inputFormat();
    delay(1000);

    PrintResult();

    delay(1000);
}

void PrintWellcomeMessage()
{
    String welcomeMessage = "Welcome to format string CTF. Try to read the
password from the stack";
    Serial.println(welcomeMessage);
    Serial.println("Enter the length of format string you want write");

    delay(1000);
}

void inputFormat()
{
    while (Serial.available() == 0) {}

    int len = Serial.parseInt();

    delay(10000);
    Serial.println("Enter your format string:");
    delay(1000);

    while (Serial.available() == 0) {}
    for (int i = 0; i < len; i++)
    {
        format[i] = Serial.read();
    }
}

void PrintResult()
{
    buff[strcspn(buff, "\n")] = '\0';
    sprintf(buff, PSTR(format));

    if (strncmp(pass, buff, sizeof(pass)))
    {
        Serial.print(buff);
        Serial.println(" does not have access");
    }

    else
    {
        Serial.println("Good job!");
    }
}
```