Exploiting stack-based buffer overflow using modern-day techniques

Stefan NICULA[1], Razvan Daniel ZOTA[1]
[1]The Bucharest University of Economic Studies
niculastefan13@stud.ase.ro, zota@ase.ro

One of the most commonly known vulnerabilities that can affect a binary executable is the stack-based buffer overflow. The buffer overflow occurs when a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. Nowadays, due to multiple protection mechanisms enforced by the operating systems, the buffer overflow has become harder to exploit. Multiple bypassing techniques are often required to be used in order to successfully exploit the vulnerability and control the execution flow of the analysed executable. One of the security features designed as protection mechanisms is Data Execution Prevention (DEP) which helps prevent code execution from the stack, heap or memory pool pages by marking all memory locations in a process as non-executable unless the location explicitly contains executable code. Another protection mechanism targeted is the Address Space Layout Randomization (ASLR), which is often used in conjunction with DEP. This security feature randomizes the location where the system executables are loaded into memory. By default, modern-day operating systems have these security features implemented. However, on the executable level, they have to be explicitly enabled. Most of the protection mechanisms, like the ones mentioned above, require certain techniques in order to bypass them and many of these techniques are using some form of address memory leakage in order to leverage an exploit. By successfully exploiting a buffer overflow, the adversary can potentially obtain code execution on the affected operating system which runs the vulnerable executable. The level of the privilege granted to the adversary is highly depended on the level of privilege that the binary is executed with. As such, an adversary may gain elevated privileges inside the system. Most of the time, this type of vulnerability is used for privilege escalation attacks or for gaining remote code execution on the system.

Keywords: *stack buffer overflow, return-oriented programming, libc attack, exploiting buffer overflow, stack protection mechanisms, address memory leak*

1 Introduction

The stack-based buffer overflow is one of the most commonly known vulnerabilities and it still one of the most exploited vulnerabilities that are affecting software and operating systems [1]. A successful exploitation of this vulnerability can sometimes be difficult to achieve and modern operating systems nowadays have protection mechanisms in place in order to prevent such issues from being exploited. These protections can also be implemented at the binary level in order to increase its security level. However, certain techniques can be used in order to bypass these prevention mechanisms but all the techniques described do need auxiliary information in order to be implemented. The study presented will be focused on Intel architecture x86, being more targeted around the Linux operating system internals and having as the main scope achieving code execution on the underlying operating system. The choice for Intel x86 architecture was being made by taking in consideration the significant difference between the x64 and x86 regarding calling conventions, general stack frame usage and registers. The x64 is far more complex compared to the x86 counterpart.

The paper will approach the exploitation of a stack-based buffer overflow by analysing the current exploitation techniques available, different protections implemented at the operating system level and on binaries. The paper continues with the analysis of bypass techniques for the aforementioned protection mechanisms, a case study that applies some of those concepts, statistics about current exploit numbers and conclusions.

2 Exploitation prevention mechanisms

The buffer overflow has been an active research topic through the history of Computer Science and multiple aspects have been addressed in order to prevent different exploitations. We can encounter multiple protection mechanisms that prevent overflow from occurring or react once the overflow happens [2].

Data Execution Prevention (DEP) is implemented at the binary level and dictates the execution privilege on a memory location. This protection prevents malicious code from being executed directly from the buffer value by allowing only specific memory locations to have execution privileges. Only certain memory blocks have execution privileges if they explicitly request so [3].

Address space layout randomization (ASLR) is implemented by the binary or by the operating system. This protection mechanism randomizes the memory address of the binary and external libraries each time it gets executed. As such, every attack which is based on static known values will fail [4].

Stack canaries/cookies assure that the stack data is not corrupted or overwritten from untrusted user-supplied data. This method works by placing a small randomly chosen value inside the program stack, in memory, just before the stack return pointer. Because the buffer overflow is writing stack memory from lower to higher address, the return pointer will be overwritten and thus the stack canary will also be modified [5].

Partial or full RELRO removes all the dynamic linked functions and ensures that the Global Offset Table (GOT) is read-only. By making GOT entries read-only, an adversary can no longer overwrite external function call addresses to a controlled stack memory address [6].

Position Independent Executable (PIE) is an optional feature that can be used at compile-time which makes the executable behaviour as a dynamic external library at linking and loading time. This feature adds more randomization in the linking and loading process. A note here is that ASLR predates PIE and ASLR does not require PIE to be enabled [7].

These protection mechanisms can prevent the exploitation of a buffer overflow and can further limit an adversary's possibilities. For most of these mechanisms, an auxiliary vulnerability that can obtain a memory leak address is mandatory in order to bypass them.

3 Exploitation techniques and protection bypasses

Exploitation techniques can vary greatly depending on each buffer-overflow case; as a result, a full exploit payload will be subjective and customized depending on the environment, the software targeted and the operating system internals. A series of protection mechanisms are presented by every single layer mentioned. From these, some have evolved into security best practices implementation, while others are still struggling to get traction. Nonetheless, we can identify some common mechanisms that can be encountered on a normal environment setup enforced with the latest default protections. This general classification will be detailed in the next sub-menus, approaching runtime protections on memory level enforced by the underlying operating system and protections implemented on the binary level.

3.1 Bypassing DEP and ASLR

Some of the most common identified protection mechanisms are the DEP and ASLR. The Data Execution Prevention mechanism is implemented at the binary level. This protection mechanism allows only specific stack frames to have execution privileges. This translates in the fact that data written in arbitrarily chosen stack-frames cannot be referenced by the instruction pointer to be executed. A good example would be a buffer overflow vulnerability that can be exploited in order to point the instruction pointer to a specific address inside the stack which is controlled by our buffer input. In this scenario, even though we have control on the Instruction Pointer, we cannot execute data that is being held in the stack frame which we overwrite. In order to bypass this protection, a technique called Return Oriented Programming (or ROP chain) can be used. By using this technique, which can also be referred to as Return to libc attack [8], we can bypass the DEP protection [9] by re-using code already present in the exploited binary. Sometimes, the studied binary does not have all the needed instructions inside its base-code in order to fully exploit an existing buffer overflow. As such, the libc attack can be used. Inside the libc library, we can re-use a variety of instructions to fulfil the scope of exploiting the vulnerability. To use the libc code-base, we need to further leak an address inside the targeted binary. This can be achieved, for example, by chaining a format string vulnerability affecting the vulnerable binary.

In regards to the format string vulnerability, this particular one is sometimes crucial in exploiting a vulnerable binary that has ASLR protection on. This is mainly due to the fact that ASLR protection is implemented by the binary or by the operating system [10]. In modern operating systems, the ASLR protection is implemented by default. As such, all the external libraries linked to the targeted binary are having randomized address values. However, the binary itself can opt to have the ASLR protection in-place. By doing this, the binary will randomize its instruction addresses and memory maps; each time the binary is executed. In order to defeat this protection mechanism, a vulnerability (information leak) such as a format string can be used in order to leak a base address that can be further used by the developed exploit. Another method of defeating this popular protection mechanism is to use a potential buffer overflow together with calling a function that uses stdout in order to print results. This can be further used by leaking GOT and PLT addresses in order to reveal libc base addresses [11]. Using the obtained libc base address from the function memory address leak, we can pinpoint the exact version of the library used by the executable. In this way, all the other function references can be calculated based on the initial libc version.

3.2 Return Oriented Programming and Return to libc attack

We can particularly note the concept of gadgets in a ROP chain. Gadgets are a set of instructions that serve our purpose of manipulating the executable in order to achieve our scope. Gadgets are pieces of code from the executable, commonly found in the loaded external libraries but can be found in the local binary code as well[12]. By using them, an adversary can do a variety of actions such as invoking syscalls while keeping the execution flow by always returning inside a stack controlled memory address. The need of RET opcode is mandatory for gadgets in order for us to keep the execution flow. Certain gadgets require different parameters that should be placed on the stack accordingly in the payload, before the function call. We know that functions are receiving parameters from the stack and because we control the stack using our buffer overflow, we can pass arguments to the called functions. In this way, we can create a chain of multiple gadgets that will provide the capabilities of executing code on the underlying operating system through the usage of the binary affected by a buffer overflow.

By comparing a typical Windows ROP chain with a Linux ROP chain, we can identify two different approaches that are quite common. For the Windows environment, oftentimes, the ROP chain's purpose is to make the stack executable and basically disable DEP using Windows API calls such as VirtualProtect, VirtualAlloc or NtSetInformationProcess whereas, for the Linux counterpart, the technique usually relies on executing directly a system command. The magic gadget from C, which basically is a code residing in the libc library that when called it's opening a shell, is typically the goto exploitation technique when using a ROP chain on a Linux binary. Of course, there are also alternatives for disabling DEP as well, on the Linux side, for example, the ret2mprotect.

3.3 Magic gadget C

As mentioned previously, one gadget that can be used to exploit a buffer overflow using ROP chain is the so-called C/C++ Magic gadget. Almost all of the libc libraries contain a version of the magic gadget. Basically, this gadget is used for ROP chaining and consists of some code residing in the libc which, when executed, opens a shell. The magic gadget code has to either call execve or issue the corresponding syscall directly. In our case, /bin/sh is set as a first argument. [13]

3.4 GOT overwrite

Another exploitation technique is defined by using the Global Offsets Table to overwrite function entries in order to execute malicious code [14]. This attack method can be avoided by implementing RELRO which basically removes all the dynamic linked functions and ensures that the GOT is read-only [15]. An example of a successful GOT rewrite would be overwriting a libc address with a local stack-frame address that contains malicious code. This can be prevented by making the GOT read-only at the initial launch of the binary file.

4 Memory leak using stdout functions

Given the constraints, aforementioned that can be applied to a specific executable, successful exploitation of a stack-based buffer overflow requires a certain memory address leak. This can be achieved in multiple ways. One of the most common techniques is finding and exploiting a format string vulnerability which basically allows us to leak values from the stack. Format string vulnerability is a type of vulnerability which allows an adversary to control the format of the printed output. [16]

Another technique that I will discuss in the next chapter is related to using certain C/C++ functions that manipulate stdout in order to leak entries from the Global Offset Table (GOT). The GOT contains the direct address of the function inside the external libraries. At compile time, that address is unknown, the dynamic linker will populate the entry when the binary is executed and the loading and linkage routines are executed. Inside the studied binary, the Procedure Linkage Table (PLT) is holding the trampoline address value to the GOT. By invoking a stdout from using the PLT address to the GOT address reference, we can obtain the actual address of a function from the loaded external library. [17]

5 Case study example

The previous chapters enumerated a series of exploitation techniques that can be implemented in order to bypass specific protection and prevention mechanisms. Let's look at the following code snippet example which is vulnerable to stack-based buffer overflow:

```
int main(){
        char local_var [60];
        puts("Enter some input:");
        fflush(stdout);
        fgets(local_var,700,stdin);
        return 0;
}
```

Inside the main function, we can note the initialization of the "local_var" variable which is of type char vector of size 60. We can note the usage of the "puts" function which we will later abuse in order to obtain an address memory leak. The "fgets" function call is receiving a stream input of max size 700 from the standard input and stores the input inside our previously declared local variable. Since there is no boundary check on the "local_var", the user can provide a size larger than the allocated size 60 of the buffer, thus resulting in a buffer overflow scenario.
For this particular exploit, we will use gdb peda and pwntools as an example of automating certain tasks and for the ease of use that these tools are bringing to the table [18]. The binary will be dynamically compiled targeting i386-x86 architecture for the Linux platform so the compiled analyzed binary will be an ELF file designed for x86 architecture.
By checking the security feature of the binary after the compilation, we can observe the following:

```
ASLR: OFF
CANARY: disabled
FORTIFY: disabled
NX: ENABLED
PIE: disabled
RELRO: partial
```

By investigating the checksec output from gdb peda, we can observe that this is a classic buffer overflow example that can be exploited using ROP chain gadgets. We can note that the binary does not have local ASLR enabled and we also note the lack of stack canaries. Even if the ASLR is disabled for the binary, the external libraries used are subject to randomization due to the ASLR enforced by the operating system [19]. We can also note that the NX (not executable) feature is enabled. By studying this particular case, we can note that the buffer overflow can be exploited but it will require a ROP chain in order to achieve code execution. That's because the NX privilege is enabled which does not allow us to redirect the execution flow in our controlled buffer but the missing stack canaries protection means that next execution instruction can be overwritten with our chosen address[20].
Considering that the binary is compiled with dynamic libraries, we will require a memory leak to obtain a base address from the libc library. Since we have no ASLR enabled on the binary level, we can search for the address of the puts function which is a stdout function. By invoking the puts using the PLT address of the puts function from the GOT, we can obtain the puts address inside the actual libc library. We want puts to call itself on the Global Offset Table which will give us the address of the puts in the binary that changes every single time [21]. We can obtain the binary PLT address of the puts by using objdump on the compiled binary.
When an external function such as puts is called, and example of a function trace call would be the following:

```
puts@plt 0x400476 -> puts@got 0x601018 -
> puts@libc linked address
```

Because the program is dynamically linked, the external libraries such as libc are resolved using PLT and GOT. The way the function trampoline works helps in this situation, the GOT entry for the puts function holds the dynamically resolved address for that specific function. The PLT contains the function trampolines to the GOT structure table. The function _dl_runtime_resolve will resolve GOT entries with the correct value for the puts function from libc.
After obtaining the puts address from the libc, we can calculate the base address of the libc itself. We need this information in order to create ROP gadgets based on libc. The library address is modified every time the binary is executed so we need to calculate its

base address in order to correctly reference other code snippets from inside the libc.
From the libc base address, we can use any specific gadget from the libc library which will provide us a reverse shell. One particular gadget described earlier is the Magic Gadget which uses a series of code snippets to execute syscall as execve into /bin/sh.[22]

6 Statistics

By analyzing the publicly available indexing measurements, we can draw some interesting conclusions based on the data collected. For example, **Fig. 1** and **Fig. 2** are showing the overall trend of the stack-based buffer overflow CVE list taken from the MITRE website from 2005 to 2019. Interestingly, the number of CVEs appears to remain constant and even increases starting from 2016 while also taking into consideration the peak reports recorded in 2007. We need to keep in mind that these are reported CVEs and they do not necessarily have a publicly available Proof of Concept or full exploit. For this data, we can refer to the exploit-db website where we can identify that all-time 321 entries are related to a stack-based buffer overflow. That means, only a handful of vulnerabilities from the ones reported annually also have publicly available exploits as well.

Year	Count	Year	Count	Year	Count
2019	114	2014	125	2009	276
2018	215	2013	107	2008	282
2017	173	2012	106	2007	394
2016	120	2011	144	2006	198
2015	134	2010	160	2005	129

Fig. 1. Stack buffer overflow CVE entries count according to MITRE

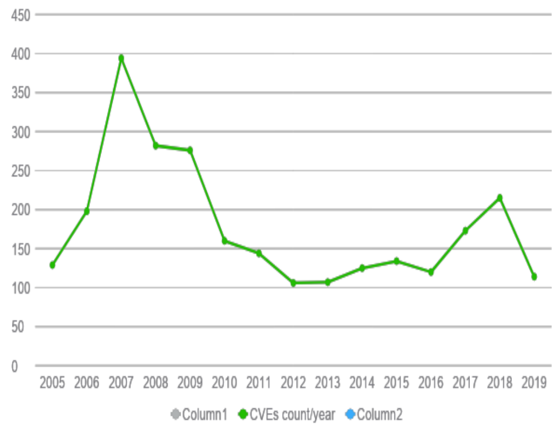

Fig. 2. Stack buffer overflow CVE entries flow chart according to MITRE

7 Edge cases and limitations

Compared to heap-based buffer overflows, the stack overflows can be considered much simpler yet they can present some interesting edge cases that are making the exploitation much harder.
Some buffer overflows could potentially be more situational than others. A good example would be the need of a partial overwrite of the EIP that is very unlikely although certain exploits do use this technique in order to bypass randomizations [24].
Limitations on exploitation can also include bad characters. Although they do not prevent the finding of the primitive buffer overflow, they are however hardening or sometimes even preventing full exploitation of the vulnerability, taking into consideration the other protection mechanisms in place as well.
Although extreme edge cases can be quite rare, full exploitation to bypass all the limitations requires a certain amount of analysis and dedication. Shellcodes can be generated while taking in consideration the bad characters as well however, the primary drawback is the size of the shellcode after the bad characters are applied. Usually, the shellcode size can exponentially get bigger with the increased number of characters to be avoided, sometimes even being impossible to generate position-independent code with too many bad characters in the blacklist [25].

Depending on the tested software, some common bad characters to be taken into consideration are 0x00, 0x0D, 0x0D and 0xFF. These characters should generally be avoided when building an exploit payload.

8 Windows vs Linux buffer overflow

On a Windows-based environment we can note specific particularities and situations when discussing stack-based buffer overflow exploitation. There are mainly two important differences that we note, we have the standard buffer-overflow that overwrites the saved returned pointer from the stack and we also have the SEH (Structure Exception Handling) based buffer overflows.

In a Windows-based software, if no explicit exception handlers are presented in the source code of the application, every thread will have an assigned handler and custom specific handlers will be added as an optional addition.[26] These values will be pushed onto the stack for each function and it will represent the pointers for treating different exceptions such as dividing by zero.

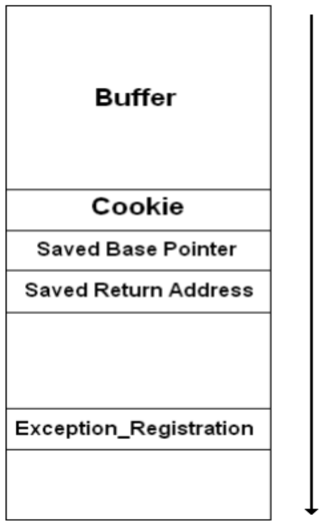

Fig. 2. Overwriting the stack with SEH entries

In a stack-based buffer overrun scenario, what would usually happen would be that the entries for the Structure Exception Handler will be overwritten by our buffer, resulting in a particular case where the return pointer is no longer our main EIP pivoting mechanism like in Linux. The SEH structure will no longer contain pointers inside their own exception handling routines but rather contain values overwritten by our buffer. This will cause the operating system to follow those values and consider them as valid addresses which would normally point to code paths that would resolve the exception.

When dealing with a SEH-based buffer overflow, a popular exploitation technique is the *pop/pop/ret* instruction set [27]. Due to the alignment on the stack for the EXCEPTION_REGISTRATION structure and the pointers associated with it, the overrun scenario often requires to pop-in two values of the stack and returning directly into our user-supplied shellcode. However, this is not always the case, depending on the situation, a SEH based buffer overflow could require multiple stack alignment moves in order to reach a known code cave.

Finally, another notable difference would be the ROP chaining creation process. Similar to the Linux case, after we take control over the EIP, we need to rely on built-in code or user-supplied shellcode in order to execute custom code on the machine. However, in our case, each Windows has a different DLL version even for the same build number, there can be differences in terms of Windows expansions, modifying the DLL version and offset itself. A hard-coded value can be used for the same deploy of Windows version but ultimately, the best approach would be a combination of memory information disclosure of a DLL base address followed by offset calculation to reach the needed function gadgets.

9 Conclusions

A stack-based buffer overflow can be exploited in multiple ways depending on a number of variables. First of all, the allocated buffer size can play a huge role in choosing the right way to exploit the issue. In the previous case study shown, the buffer size was generous and allowed us enough space to inject various addresses and use multiple techniques without worrying too much about the memory space. If the buffer size was restricted to a small limited number of characters, additional steps would be required to successfully exploit the vulnerability. For example, an additional input buffer may have been required to redirect the execution flow into it however, that hypothetical input needed to be, again, controlled by the adversary. Some techniques used to get around the limited buffer size promote the usage of environmental variables that are loaded by the binary when executed. A memory leak address is needed in order to obtain such details. The second problem raised is related to the protection mechanisms that are preventing a straight forward exploitation technique. We cannot redirect the execution flow directly into our defined buffer due to DEP. Also, the address to libc functions is randomized each time we execute the binary given the ASLR protection enabled. We also have a disabled RELRO which should allow us some opportunities to overwrite the GOT-PLT entries inside the memory blocks. In order to bypass the aforementioned protections, a memory address leak is mandatory in order to obtain an address so we can further calculate our needed function addresses. A stack-buffer overflow cannot be exploited stand-alone, it can be situational and certain memory leak vulnerabilities are required given the protection mechanisms encountered in the process. A very important role is how to understand the internals of a program as well as properly identifying and using external libraries loaded by the executable in order to achieve code execution.

Buffer overflows are still emerging, active and real threats. Yearly, this specific vulnerability can be encountered in multiple CVEs reported on popularly known software [23]. In order to successfully exploit them, certain techniques are required in order to bypass common protection mechanisms. Nevertheless, these vulnerabilities are still found in solutions that have a high level of maturity in terms of security best practices and implementations. We should not overlook nor undermine their potential risk, even though modern-day systems are implementing multiple protection mechanisms in order to try and prevent such attacks.

References

[1] National Institute of Standards and Technology. ICAT Metabase.http://icat.nist.gov/

[2] Erick Leon, Stefan D. Bruda, Counter-measures against stack buffer overflows in GNU/Linux operating systems., The International Workshop on Parallel Tasks on High Performance Computing, Procedia Computer Science 83, 2016, Volume 83, pages 1301 – 1306

[3] A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, (Jul.2017) https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in, retrieved Dec.2018

[4] Address Space Layout Randomization, (Mar.2003), https://pax.grsecurity.net/docs/aslr.txt, retrieved Feb. 2015.

[5] Buffer overflow protection, (Jun.2018), https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries, retrieved Jan.2019

[6] Hardening ELF binaries using Relocation Read-Only (RELRO), (Jan.2019), https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro, retrieved Jan.2019

[7] Position Independent Executables (PIE), (Nov.2012), https://access.redhat.com/blogs/766093/posts/1975793, retrieved Jan.2019

[8] Return-to-libc Exploit, (Feb.11), https://medium.com/@nikhilh20/return-to-libc-exploit-aa3fe6fb0d69, retrieved Mar.2019

[9] Bypassing DEP with ROP (32-bit), (Dec.2017), https://bytesoverbombs.io/bypassing-dep-with-rop-32-bit-39884e8a2c4a, retrieved Mar.2019

[10] Bypassing ASLR - Part I, (May 2015), https://sploitfun.wordpress.com/2015/05/08/bypassing-aslr-part-i/, retrieved Dec.2018

[11] Yan Fen, Yuan Fuchao, Shen Xiaobing, Yin Xinchun, Mao Bing, A New Data Randomization Method to Defend Buffer Overflow Attacks, International Conference on Applied Physics and Industrial Engineering, Physics Procedia 24, Volume 24, Part C, 2012, pages 1757-1764

[12] Bruce Dang , Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation, Wiley Publishing, 2014

[13] The magic gadget, (Sep.2016),)https://github.com/m1ghtym0/magic_gadget_finder, retrieved Apr.2019

[14] How to hijack the Global Offset Table with pointers for root shells, (Apr.2006), https://www.exploit-db.com/papers/13203, retrieved Apr.2019

[15] Ryan "elfmaster" O'Neill, Learning Linux Binary Analysis, Packt, 2016

[16] Format String Exploitation-Tutorial, https://www.exploit-db.com/docs/english/28476-linux-format-string-exploitation.pdf, retrieved Apr.2019

[17] P. Silberman and R. Johnson, A Comparison of Buffer Overflow Prevention Implementations and Weaknesses, presentation at Black Hat USA, Caesar's Palace, Las Vegas, NV, USA (Jul. 2004).

[18] Eldad Eilam, Reversing: Secrets of Reverse Engineering, Wiley Publishing, 2005

[19] Sahel Alouneh, Mazen Kharbutli, Rana AlQurem, Stack Memory Buffer Overflow Protection Based on Duplication and Randomization, The 4th International Conference on Emerging Ubiquitous Systems and Pervasive Networks, Procedia Computer Science 21, 2013, pages 250 – 256

[20] G. Duarte. Epilogues, Canaries, and Buffer Overflows, (Mar. 19 2014), http://duartes.org/gustavo/blog/post/epiloguescanaries-bufferoverflows/ , retrieved Feb. 2015.

[21] Ryan "elfmaster" O'Neill, Learning Linux Binary Analysis, Packt, 2016

[22] Smashing the Stack, (Apr.2014), http://phrack.org/issues/49/14.html, retrieved Oct.2018

[23] Mitre CVE Buffer Overflow search result, https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Buffer+Overflow, retrieved May.2019

[24] Kai Jander, Lars Braubach , Alexander Pokahr, Practical defense-in-depth solution for microservice systems, International Journal of ubiquitous systems and pervasive networks (JUSPN), volume 11, issue 1, 2019, pages 17-25

[25] Madjid Kara, Olfa Lamouchi, Amar Ramdane-Cherif, Software quality assessment algorithm based on fuzzy logic, International Journal of ubiquitous systems and pervasive networks (JUSPN), volume 8, issue1, 2017, pages 01-09

[26] Defeating the Stack Based Buffer Overflow pevention mechanism of Microsoft Windows 2003 Server, BlackHat Asia 03(Sept.2003), https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf, retrieved Aug.2018

[27] The need for a POP POP RET instruction sequence, (Oct.2010),

https://dkalemis.wordpress.com/2010/10/27/the-need-for-a-pop-pop-ret-instruction-sequence/, retrieved Oct.2019

Stefan NICULA graduated from the Faculty of Cybernetics, Statistics and Economic Informatics of the Bucharest University of Economic Studies in 2016 and followed a Master's degree in IT&C Security at the same university. He is a threat researcher and pentester with over 5 years of experience. His areas of expertise are in penetration testing, malware analysis, reverse engineering, and exploitation techniques, with a passion for Windows internals, vulnerability research, exploit development, and mitigation techniques. At present he is pursuing a PhD in Information Security at the Bucharest University of Economic Studies, focusing on heap memory exploits on browsers, Windows kernel vulnerabilities and fuzzing Windows API functions. Current publications and public presentations held by Stefan are covering areas such as IoT security evaluation and Windows binary exploitation, latest malware trends and recent developments in the exploit development field.

Răzvan Daniel ZOTA has graduated the Faculty of Mathematics – Computer Science Section at the University of Bucharest in 1992. He has also a Bachelor degree in Economics and a postgraduate degree in Management from SNSPA Bucharest, Romania. In 2000 he has received the PhD title from the Academy of Economic Studies in the field of Cybernetics and Economic Informatics. From 2010 he is supervising PhD thesis in the field of Economic Informatics, part of the Doctoral School of Economic Informatics in the Bucharest University of Economic Studies.