A Taxonomy of Defenses against Memory Corruption Attacks

Bojan Novković
Department of Electronics, Microelectronics, Computer and Intelligent Systems
University of Zagreb, Faculty of Electrical Engineering and Computing
Zagreb, Croatia
bojan.novkovic@fer.hr

***Abstract*—Vulnerabilities caused by memory corruption related bugs are a pervasive threat, continually undermining the security of the whole computing environment. The lack of memory safety mechanisms in indispensable systems programming languages like C or C++ leaves plenty of room for programmer-induced errors which often result in catastrophic security breaches.**

We analyze the root causes of these vulnerabilities, providing a concise overview of memory corruption bugs and related attacks. Although present in a number of programming languages, we use the C/C++ programming language as a basis for our analyses. We categorize existing defensive mechanisms based on the attack techniques they focused on preventing and give a brief insight into state-of-the-art defenses introduced throughout the years, with a special focus on operating system defenses.

Keywords*—*memory corruption, systems security, vulnerabilities

I. Introduction

Despite being one of the most extensively studied problems in computer security, memory corruption bugs still remain among one of the most common attack vectors. According to a recent ENISA technical report [1], four out of the top ten most common software weaknesses and seven out of the top ten critical software vulnerabilities are related to memory corruption bugs. A majority of memory corruption bugs are present in systems programming languages, mainly in C and C++. This fact stems from the lack of builtin memory safety validation mechanisms, as the design of these languages focused on sacrificing safety for efficiency and speed. In practice, entrusting the often gargantuan task of memory safety validation to the programmer routinely produces bugs which are used for mounting attacks on software.

This paper aims to provide an insight into the endless arms race present in research of defense mechanisms against memory corruption attacks along with a survey of the current state-of-the art techniques.

II. Memory corruption vulnerabilities

Since the seminal work on subverting execution flow via stack manipulation [2] (commonly referred to as *stack smashing*), a whole new field of research has emerged, leading to numerous discoveries on abusing, preventing and mitigating memory corruption bugs. Despite the immense research efforts in this field, vulnerabilities caused by memory related bugs still plague modern software as attacks exploiting these vulnerabilities continue to grow more sophisticated. We focus on enumerating vulnerabilities specific to the C/C++ programming language and the `x86_64` ISA.

A. Memory safety violation

Memory safety violations are the most severe class of vulnerabilities and have been extensively studied [3], [4]. Although intuitively understood, the term ”memory safety” lacks a unified definition and is often not thoroughly defined in research. A common approach in defining memory safety takes safe program execution into consideration. A program is deemed to be memory safe if no common, well known memory errors can occur during its execution. Song *et al.* [4] provide a more detailed definition of memory safety by defining the *intended referents* of pointers. A program is then deemed to be memory safe if, during its execution, all pointers point to their *intended referents* while those referents are valid.

1) Temporal safety violation: Temporal safety violations occur when a pointer to a memory block which is no longer valid (*dangling*) is accessed. Accessing a dangling pointer is commonly known as a *use-after-free* vulnerability. These types of violations usually manifest themselves in the form of a program crash, but in the presence of a malicious actor who can reuse the freed object they can lead to arbitrary code execution and data corruption.

2) Spatial safety violation: Spatial safety violations occur when a program dereferences a pointer to an out-of-bounds memory location. These type of memory corruption vulnerabilities are one the most studied and devastating type of vulnerabilities which, if exploited properly, can provide the attacker with arbitrary code execution. The most well-known representative of these vulnerabilities is the notorious *buffer overflow* vulnerability, which constitutes of writing to memory beyond the intended buffer, both on the stack and the heap. It is commonly caused by mismatches between the buffer size and the size used when writing to the buffer, as depicted in Listing 1.

```
static int nfs_readlink_reply(uchar *pkt, unsigned
    len) {
    struct rpc_t rpc_pkt;
    [...]
        memcpy((unsigned char *)&rpc_pkt, pkt, len);
    [...]
}
```

Listing 1. A stack buffer overflow vulnerability found in **CVE-2019-14204**. User provided packet data was copied without checking the provided length parameter.

B. Abuse of variadic functions

Variadic functions are functions which take an arbitrary number of arguments. In C/C++, support for variadic functions is provided by the compiler itself or manually implemented by the programmer. Unfortunately, in the presence of unsanitized user input, variadic functions can often be manipulated in a way which leads to spatial memory violations, type mismatch errors and undefined behaviour, depending on the implementation of the vulnerable function. Moreover, since function arguments are passed using the stack, malicious actors who are able to manipulate the variadic functions can leak sensitive contents from the stack which can be subsequently used to bypass existing defenses [5].

C. Type confusion

Type confusion is a memory corruption bug found in programming languages with lax type systems. It arises from unsafe casting between polymorphic classes which can lead to the *virtual table* pointer corruption, possibly leading to code injection or spatial safety violation [6]. For instance, the C++ programming language provides several ways of performing explicit type conversion [7], but only the `dynamic_cast` conversion performs additional type checks, albeit constrained to polymorphic classes only, and incurs additional run-time overhead. Consequently, it is often abandoned in favor of the `static_cast` conversion which is performed at compile time. Unfortunately, this compile time check does not hold during run-time where the observed type could deviate from the underlying memory contents [6].

D. Undefined behaviour

The C++ language standard defines undefined behaviour as a ”behavior for which this International Standard imposes no requirements.” [7]. In such cases compiler designers assume that the programmer will not write code which exhibits such behaviour, which allows them to generate more appropriate code for the target architecture [8]. Undefined behaviour constructs are therefore subject to aggressive compiler optimizations which often cause deviations from the expected and reproduced program behaviour and in some cases lead to serious vulnerabilities.

1) Improper initialization of variables: Upon declaration and before initialization, variables in C/C++ contain an *indeterminate* value, as the compiler is not required to initialize the variable to a default value [7]. Uninitialized variables often contain leftover and stale data from previously used memory, both on the heap and the stack. As shown in Listing 2, use of these variables can result in control flow violation, spatial and temporal safety violations and leakage of sensitive information.

2) Signed overflow: Integer overflows are caused by unsafe conversion between signed and unsigned integer types. Moving a signed integer to an unsigned integer variable can cause integer overflow and result in a huge value. Attackers can take advantage of these vulnerabilities to violate spatial memory safety and mount attacks.

3) Compiler-induced vulnerabilities: Compiler optimizations can convert undefined behaviour bugs into full blown vulnerabilities [8]. By assuming that the source code is always free of undefined behaviour, compiler optimizations can omit security checks. An example of this would be the compiler optimizing away a part of code which checks whether a pointer variable contains a `NULL` value [8].

```
int ParseWave64HeaderConfig (FILE *infile, char *
    infilename, char *fourcc, WavpackContext *wpc,
    WavpackConfig *config) {
    [...]
    WaveHeader WaveHeader;
    [...]
    if (!WaveHeader.NumChannels)
    [...]
    total_samples = (infilesize - DoGetFilePosition
    (infile)) / WaveHeader.BlockAlign;
    [...]
}
```

Listing 2. An abridged version of a vulnerability caused by the use of an uninitialized variable, found in **CVE-2019-1010319**.

III. Attack techniques

A. Code injection

Spatial and temporal memory safety violations can be leveraged to inject and execute code into the memory space of a process by hijacking the control flow [9]. Such attacks usually exploit the buffer overflow vulnerability to inject malicious code, often referred to as *shellcode* [9], into the stack or heap, attempting to execute it. This class of attacks has been largely curbed by defensive policies found in modern operating systems.

B. Code reuse

Rather than injecting new code into the process, code reuse attacks leverage spatial and temporal memory safety violations to chain existing pieces of code, called *gadgets*, to achieve arbitrary code execution. The first known code reuse attack was published in 1997 under the name *return-into-libc* [9]. It relies on specific C standard library functions and exploits vulnerable programs without code injection by rewriting the return address to point to a specific C standard library function.

1) Return Oriented Programming: Code reuse attacks soon proved to be much more dangerous when Krahmer [10] showed that it was possible to omit function calls from code reuse attacks, replacing them with short sequences of assembly code. Shortly after, Shacham [11] published his seminal work on code reuse attacks, showing that code reuse attacks could perform arbitrary computation on the `x86` ISA by chaining existing assembly code sequences. The technique was named *Return-Oriented Programming (ROP)* [11].

Similar to code injection attacks, ROP relies on control-flow hijacking via manipulating return addresses found on the stack, but differs in the way computation is achieved. After finding an appropriate spatial memory safety violation, an attacker searches the executable file for *gadgets* and notes their addresses. She can then form a payload which mimics a stack, consisting of a series of return addresses pointing to previously

found gadgets. After exploiting the vulnerability, the sequence of gadgets is executed. It should be noted that the payload is also responsible for overwriting the appropriate stack pointer register to point to the payload. This is commonly referred to as *stack pivoting* [9].

2) Data Oriented Programming: Spatial memory safety violations can also be leveraged to corrupt local function variables present on the stack. This type of attack is referred to as a *Data-oriented attack* [12]. This class of attacks was recently shown to be even more threatening as Hu *et al.*'s [13] seminal work proved that *data-oriented attacks* can be used to construct arbitrary programs. This novel technique, named *Data Oriented Programming (DOP)*, utilizes code gadgets in a fashion similar to ROP attacks, with a key difference in the way computation is achieved. DOP attacks rely on *gadget-dispatchers*, existing code pieces which are used to chain separate gadgets in an arbitrary sequence [13]. However, DOP gadgets are used to *simulate* operations through executed instructions while preserving the original execution flow, completely bypassing a widely used defense against code-reuse attacks.

A vulnerable code snippet is shown in Listing 3, where several stack variables are susceptible to malicious manipulation via overflowing the adjacent buffer. The attacker is thus able to use the `while` loop as a *gadget-dispatcher*, chaining highlighted gadgets present in the loop body by manipulating the value of variables used in conditional branching statements as well as the value of operands in some expressions.

```
int control_variable = <arbitrary value>;
short* tag; int* msg_seq_num;
char msg_buffer[BUFSIZE];
tag = &msg_buffer[TAG_OFFSET];
seq_num = &msg_buffer[SEQNUM_OFFSET];

while(control_variable--){
    // buffer overflow
    read(user_fd, msg_buffer, user_data_size);
    if(*tag == 0){ // controllable condition
        // controllable dereference
        *tag = *msg_seq_num;
    }
    else{
        // controllable addition
        *tag = *msg_seq_num + control_variable;
    }
}
```

Listing 3. A simplified example of a DOP attack.

IV. Current State of Defenses

A vast number of countermeasures aimed at thwarting attacks caused by memory corruption vulnerabilities have been implemented over the last two decades. Although a majority of countermeasures incorporate multiple modifications of various areas in the whole computing environment, we classify them based on the attack techniques they focused on the most.

A. Countering specific attack techniques

Code injection attacks have been largely defeated with lightweight mitigations. *Stack canaries*, a widely deployed and successful mitigation against stack-based buffer overflows, have been initially proposed by several researchers. The technique verifies the integrity of the current stack frame by placing a random value (known as a *canary* or *cookie*) before the return address, checking its value in each function epilogue. This technique has proven to be extremely effective and was applied to various other mitigations. Unfortunately, *stack canaries* are vulnerable to information leaks and brute-force attacks. Recent research on canaries focused on leveraging novel hardware features to strengthen canaries [14] and applying dynamic polymorphism to existing schemes to thwart the aforementioned attacks [15]. *Shadow stacks*, a mechanism which uses an additional, protected stack to verify the integrity of return addresses, have also been proposed in lieu of *stack canaries* [3]. Another set of countermeasures focused on extending the paging memory management model to ensure that all stack and data pages are not simultaneously executable and writable. Combined with *stack canaries*, they offer complete protection against code injection attacks.

TABLE I
Classification of existing defense mechanisms

Defense mechanism	Code injection	ROP	DOP
W^X/DEP	✓	✗	✗
Stack canaries	✓	✧	✗
Shadow stacks	✓	✓	✗
ASLR	✓	✧	✗
KASLR	✓	✧	✗
CFI	✓	✓	✗
Fine-grained code randomization	✓	✓	✓
Statistical detection	✗	✓	✓

✓ Effective ✗ Ineffective ✧ Partial mitigation

ROP attacks are an ever-present threat despite numerous defense mechanisms hardening their execution. *Address Space Layout Randomization* was among the first proposed countermeasures against code reuse attacks, focusing on thwarting ROP attacks by removing deterministic placement of gadgets through randomizing parts of a process's memory layout. It has seen widespread adoption and proved to be efficient in hardening ROP attack execution. However, it is also a subject of critique as existing implementations exhibit vulnerabilities to memory disclosure and brute-force attacks due to insufficient entropy [9].

An effective countermeasure which leverages execution-flow deviations caused by code-reuse attacks was proposed in 2005. by Abadi et al. [16]. Dubbed as *Control Flow Integrity (CFI)*, it aims to restrict the set of possible execution flow transfers to precomputed, valid ones. A graph containing all valid changes in control flow, the so called *Control Flow Graph (CFG)* is computed before the target program is executed, and is used at run-time to enforce valid execution flow transfers through inlined checks before each execution flow transfer [16]. However, due to the high computational overhead of precise control-flow graph computing, initial implementations of CFI defenses relied on coarse-grained graphs combined with more lenient policies. Unfortunately, Carlini *et al.* [17]

demonstrated that these defense schemes can be bypassed in a simple manner, introducing a variation of the ROP paradigm named *Call Oriented Programming*. Recent research in CFI defenses focused on improving security guarantees by improving previously used contextual information and increasing performance by leveraging underlying hardware features [18]. Moreover, recently announced hardware features such as Intel's *Control-Flow Enforcement* or ARM's *Branch Target Identification* open new frontiers in research of hardware-backed CFI systems.

A significant amount of research was also focused on efficient run-time detection of ROP attacks, often relying on information found in *Hardware Performance Counter* registers present in recent CPUs. Early systems combined coarse-grained CFI with heuristics based on recent execution flow, terminating the program if one of the several criteria for ROP-like behaviour was met. However, the execution flow history inspection approach was soon proved to be bypassable [17]. Several researchers noted that ROP payloads introduce branch mispredictions for each gadget used in the payload, which, combined with several metrics regarding execution, paved the way for systems which detect ROP attacks and payloads using statistical inference. A multitude of systems utilizing learning-based statistical methods for statically detecting ROP payloads in files have also been proposed [19].

DOP attacks are a new, rising threat which completely bypasses widely used defenses against code-reuse attacks. Although this technique was only recently introduced, it quickly sparked research interest, resulting in many novel applications and defenses. Although widespread defenses like CFI are largely bypassed by this class of attacks, existing fine-grained randomization schemes such as *Data Space Randomization (DSR)* can curb execution of such attacks [5]. Rajasekaran et al. [5] revitalized and upgraded the idea behind *DSR*, showing that its existing design is easily bypassed, and proposed a novel data rerandomization scheme which supports continuous and on-demand randomization of data present in a process, thwarting existing DOP attacks [5]. Following the notion of thwarting DOP attacks through data randomization, Aga et al. [20] proposed *Smokestack*, a system which randomizes the stack layout of functions upon calling them at run-time, with an acceptable overhead.

Several run-time detection systems have also been proposed. Following a previously well established notion of inspecting micro-architectural state, Cheng *et al.* [21] proposed a system which leverages statistical inference to perform lightweight monitoring by detecting anomalies in control flow transfers. Xiao *et al.* [22] evaluated the threat posed by DOP attacks in the context of operating system kernels and proposed a monitoring system which ensures integrity of kernel data. They took the approach of splitting all kernel data into four categories depending on the frequency of their modification, using them to monitor and share data with peers in order to detect possible anomalies [22].

B. Operating system defense mechanisms

All modern operating systems use a variety of hardware-assisted mechanisms for enforcing a strict separation policy to prevent malicious software from wreaking havoc on the whole system, with the basic line of defense consisting of appropriate memory management models and ring-based privilege separation. The paging memory management model has proved to be efficient in enforcing separation while providing flexible integration for other defense mechanisms [23].

However, the advent of code injection attacks prompted the development of new defensive policies which focused on modifying the memory management model. In 2003., the OpenBSD project introduced their novel *WˆX* policy [24] which thwarts code injection attacks by ensuring that no mapped page is simultaneously marked as both executable and writable. Shortly thereafter, Microsoft announced a similar, hardware-backed policy called *Data Execution Prevention (DEP)* [23]. This policy took advantage of newly released changes in the `x86` page table entry layout which now included a designated bit controlling the executable property of a page, known as the NX bit [23].

Although DEP/*WˆX* combined with stack integrity protection mechanisms successfully prevents the majority of code injection attacks, those defensive policies are completely circumvented by code reuse attacks. Prompted by the threat from *return-to-libc* attacks, a novel lightweight defense mechanism focused on virtual memory permutations was introduced in 2001. by the PaX project [25]. Named *Address Space Layout Randomization (ASLR)*, it randomizes the virtual memory layout of a process, usually at its creation. Since code reuse attacks rely on jumping to specific addresses inferred from executable files, introducing non-deterministic placement of code in memory significantly complicates attack execution, theoretically leaving brute-force gadget address guessing as the only possible option. Implementations of ASLR differ between operating systems, ranging from randomizing base offsets of all process areas to complete system-wide randomization of offsets and placement for all major components, from the kernel to all process elements. However, the rise of ROP techniques soon showed that a partial application of ASLR offers no security guarantees as every non-randomized component increases the attack surface. Moreover, even a full application of ASLR can be circumvented by disclosure of memory via information leak vulnerabilities as leaked memory contents may provide enough information to completely derandomize parts or the whole memory layout of a process [25].

ASLR has proven to be a fruitful research topic yielding a multitude of attacks on specific implementations, designed to completely bypass all ASLR security properties, ranging from information leak vulnerabilities to side-channel attacks [25]. Consequently, a significant amount of existing research was also focused on hardening ASLR by mitigating information leaks, usually focusing on increasing randomization entropy to invalidate potential leaks in a timely manner [26], [27].

Chen *et al.* [26] introduced a system which aims to raise randomization entropy by performing compiler assisted on-demand run-time re-randomization of executable code. They tackled the preservation of code validity through intra-function randomization of basic assembly code blocks, which augmented performance while preserving added security benefits [26]. Williams-King *et al.* further developed run-time rerandomization by introducing a defense technique which leverages continuous rerandomization of code locations, requiring no modifications to the computing environment [27]. Research in fine-grained randomization also yielded a number of proposed defenses which randomize process data. In addition to making exploitation of memory vulnerabilities harder, some proposed systems offer invaluable detection of memory corruption bugs during development [4].

Over the course of the last two decades, ASLR had a number of novel applications on existing systems. Although originally intended for userspace components, Microsoft introduced *Kernel Address Space Layout Randomization (KASLR)* in 2007. , adding a significant hurdle for kernel exploitation. However, KASLR is often criticized its ineffectiveness in thwarting attacks along with insufficient entropy, and was broken several times through cache attacks, side channel attacks and abuse of underlying CPU features. Nonetheless, recent research in this area suggests that fixing KASLR through improving software-based isolation is possible and viable [28].

An additional approach for increasing kernel entropy was proposed and implemented by the OpenBSD project in 2017. Instead of randomizing base offsets for kernel code, their *kernel address randomized link (KARL)* technique randomizes all objects which are linked into the kernel binary at boot, resulting in a unique kernel layout every time the operating system is booted [29].

Privilege separation plays a key role in defending against memory corruption attacks. Recent developments in privilege separation enforcing mechanisms have been facilitated by hardware supported features. Vahldiek-Oberwagner *et al.* [30] have recently proposed *ERIM*, a system providing efficient in-process isolation using Intel's *Memory Protection Keys* hardware feature. Acting as an intermediary between the userspace process and the kernel, it ensures that a malicious attacker cannot alter the existing MPK protection scheme by designating special *call gates* which transfer control between trusted and untrusted process components. The system intercepts relevant system calls, scanning all trusted executable pages for code violating the aforementioned constraint before they are mapped [30].

Development was also made in software-based isolation and privilege separation techniques. The OpenBSD project recently introduced a new operating system mechanism called *pledge* designed to facilitate enforcement and adoption of privilege separation. The mechanism defines subsets of system calls which are passed to the *pledge* system call, causing abrupt program termination if, later on, a system call which was not registered is called [31]. By adopting a system-call based approach instead of complex, domain-specific languages, *pledge* can be conveniently integrated into existing source code.

Defenses for code executing in kernel memory space have also been proposed. Gens et al. [32] have noted that crucial parts of an operating system kernel are subject to data-oriented attacks, demonstrating an attack which corrupts page table entries in a hardened Linux kernel. As a countermeasure to this specific attack, they presented *PT-Rand*, a defense mechanism which utilizes randomization to protect all pages used as page tables, converting physical addresses to virtual ones at run-time [32].

C. Secure memory allocation

V. Conclusion

Despite a great number of efficient and effective defenses against memory corruption vulnerabilities developed in recent decades, attacks abusing these vulnerabilities are still a persistent threat. The life cycle of introduced countermeasures can be brief. Once subjected to thorough analysis or when faced with a new attack paradigm, novel defense mechanism can be completely invalidated. This hectic pace combined with a narrow focus on thwarting very specific attacks can hinder adoption of defense mechanisms in existing systems. In extreme cases, even applied defense mechanisms can be inadvertently disabled by other actors in the computing environment [33]. All-encompassing defenses such as *DEP* have stood the test of time and have become a security standard, demonstrating that mechanisms focused on the foundations of the computing environment can significantly improve systems security.

With the recent emergence of the DOP attack paradigm, existing defense mechanisms are being re-evaluated and new ones proposed in their stead. In spite of existing research efforts, crucial components of the computing environment such as operating system kernels are still subject to these novel attacks, undermining existing security guarantees.

Acknowledgment

This work has been carried out within the *AIPD2, Digital platform for personal data lifecycle management protection* project, funded by the European Regional Development Fund.

References

[1] V. Katos, S. Rostami, P. Bellonias, N. Davies, A. Kleszcz, and S. F. et al., "State of Vulnerabilities 2018/2019: Analysis of Events in the life of Vulnerabilities," The European Union Agency for Cybersecurity, Tech. Rep., Dec. 2019. [Online]. Available: https://www.enisa.europa.eu/publications/technical-reports-on-cybersecurity-situation-the-state-of-cyber-security-vulnerabilities/at_download/fullReport.

[2] O. Aleph, "Smashing the stack for fun and profit," 1996. [Online]. Available: http://www.shmoo.com/phrack/Phrack49/p49-14.

[3] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, IEEE, 2013, pp. 48–62.

[4] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for Security," *CoRR*, vol. abs/1806.04355, 2018. eprint: 1806.04355.

[5] P. K. Rajasekaran, "Practical Run-Time Mitigations Against Data-Oriented Attacks," Ph.D. dissertation, UC Irvine, 2020.

[6] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. Van Der Kouwe, "TypeSan: Practical type confusion detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 517–528.

[7] ISO, *ISO/IEC JTC1 SC22 WG21 N3690: Working Draft, Standard for Programming Language C++. May 2013.* Geneva, Switzerland: International Organization for Standardization, 2013, p. 1214. [Online]. Available: https://isocpp.org/files/papers/N3690.pdf.

[8] M. J. Hohnka, J. A. Miller, K. M. Dacumos, T. J. Fritton, J. D. Erdley, and L. N. Long, "Evaluation of Compiler-Induced Vulnerabilities," *Journal of Aerospace Information Systems*, vol. 16, no. 10, pp. 409–426, 2019.

[9] L. V. Davi, "Code-reuse attacks and defenses," Ph.D. dissertation, Technische Universität, 2015.

[10] S. Krahmer, *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*, 2005.

[11] H. 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*, 2007, pp. 552–561.

[12] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. Yao, "Exploitation techniques and defenses for data-oriented attacks," in *2019 IEEE Cybersecurity Development (SecDev)*, IEEE, 2019, pp. 114–128.

[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 969–986.

[14] H. Liljestrand, Z. Gauhar, T. Nyman, J.-E. Ekberg, and N. Asokan, "Protecting the stack with paced canaries," in *Proceedings of the 4th Workshop on System Software for Trusted Execution*, 2019, pp. 1–6.

[15] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao, "To detect stack buffer overflow with polymorphic canaries," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2018, pp. 243–254.

[16] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[17] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 385–399.

[18] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 195–211.

[19] T. Usui, T. Ikuse, Y. Otsuki, Y. Kawakoya, M. Iwamura, J. Miyoshi, and K. Matsuura, "Ropminer: Learning-based static detection of rop chain considering linkability of rop gadgets," *IEICE Transactions on Information and Systems*, vol. 103, no. 7, pp. 1476–1492, 2020.

[20] M. T. Aga, "Thwarting advanced code-reuse attacks," Ph.D. dissertation, 2019.

[21] L. Cheng, H. Liljestrand, T. Nyman, Y. T. Lee, D. Yao, T. Jaeger, and N. Asokan, "Exploitation Techniques and Defenses for Data-Oriented Attacks," *CoRR*, vol. abs/1902.08359, 2019. eprint: 1902.08359.

[22] J. Xiao, H. Huang, and H. Wang, "Kernel data attack is a realistic security threat," in *International Conference on Security and Privacy in Communication Systems*, Springer, 2015, pp. 135–154.

[23] S. Andersen and V. Abella, *Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies*, 2004.

[24] T. DeRaadt, "Advances in OpenBSD," *CanSecWest, Vancouver, Canada*, 2003.

[25] H. Marco-Gisbert and I. Ripoll, "On the Effectiveness of Full-ASLR on 64-bit Linux," in *Proceedings of the In-Depth Security Conference*, 2014.

[26] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the sixth ACM conference on data and application security and privacy*, 2016, pp. 50–61.

[27] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 367–382.

[28] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," in *ASIA CCS 2020-Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ACM/IEEE, 2020.

[29] J. Edge, *OpenBSD kernel address randomized link*, 2017. [Online]. Available: https://lwn.net/Articles/727697/.

[30] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "{ERIM}: Secure, Efficient In-process Isolation with Protection Keys ({MPK})," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1221–1238.

[31] J. Anderson, "A comparison of unix sandboxing techniques," *FreeBSD Journal*, 2017.

[32] D. Gens, "Os-level attacks and defenses: From software to hardware-based exploits," Ph.D. dissertation, Technische Universität, 2019.

[33] X. Ge, M. Payer, and T. Jaeger, "An evil copy: How the loader betrays you.," in *NDSS*, 2017.