Detection, Exploitation and Mitigation of Memory Errors

Oscar Llorente-Vazquez[1], Igor Santos[2,3], Iker Pastor-Lopez[1], and Pablo Garcia Bringas[1]

[1] University of Deusto, Bilbao, Spain
{oscar.llorente, iker.pastor, pablo.garcia.bringas}@deusto.es,
[2] Mondragon Unibertsitatea, Arrasate-Mondragon, Spain
isantos@mondragon.edu
[3] HP Labs, Bristol, United Kingdom
igor.santos.grueiro@hp.com

Abstract. Software vulnerabilities are the root cause for a multitude of security problems in computer systems. Owing to their efficiency and tight control over low-level system resources, the C and C++ programming languages are extensively used for a myriad of purposes, from implementing operating system kernels to user-space applications. However, insufficient or improper memory management frequently leads to invalid memory accesses, eventually resulting in memory corruption vulnerabilities. These vulnerabilities are used as a foothold for elaborated attacks that bypass existing defense methods. In this paper, we summarize the main memory safety violation types (i.e., memory errors), and analyze how they are exploited by attackers and the main mitigation methods proposed in the research community. We further systematize the most relevant techniques with regards to memory corruption identification in current programs.

Keywords: Memory errors, memory safety, memory corruption

1 Introduction

Memory and type unsafe languages such as C and C++ do not ensure that memory accesses in programs are valid, which can lead to program bugs that result in memory corruption and eventual exploitation, compromising system security. Over the years, memory errors have been detected in countless programs of any type and nature [56, 12], and have since established themselves as the most dangerous and exploited vulnerabilities [13].

In spite of massive advances in memory safety violations detection and mitigation, memory errors persist in current versions of software, as they are further being exploited to carry out increasingly sophisticated attacks [21, 17]. Unfortunately, a significant part of the proposed approaches are not adopted in real-world scenarios, either because of performance concerns, compatibility issues, or insufficient mitigation capabilities [51]. In fact, a previous work [55] analyzed the

evolution of the memory corruption ecosystem, and concluded that it is unlikely that memory errors will become less important in the near future, a fact that remains true at present.

More worryingly, memory corruption does not take place exclusively during sequential execution. Depending on the interactions between threads and the scheduling, new memory errors can arise in concurrent programs, leading to concurrency memory corruption vulnerabilities that traditional detection tools fail to unmask [29, 6].

In this paper, we aim at bringing a broad comprehension into the field of memory corruption. To this end, we thoroughly describe memory safety, review existing memory corruption identification techniques, and expand on further attacks as well as mitigation approaches.

The rest of the paper is organized as follows. Section 2 introduces memory safety and enumerates memory safety violations, organizing them on the basis of their characteristics. Section 3 presents the wide variety of exploitation methods around memory errors along with the most relevant mitigation strategies, with an emphasis on sophisticated attacks such as those that reuse code already present in memory, and those that target non-control data. Then, section 4 categorizes and details the main methods used to detect memory errors. Section 5 surveys analogous research works in the literature. Lastly, section 6 presents the conclusions of the paper.

2 Memory Safety

Explicit memory management and pointer arithmetic are convenient features of the C and C++ programming languages that make them the ideal choice for many tasks. Their implementations enable developers to manage and access objects stored in memory manually, providing static, automatic, and dynamic memory allocation (and deallocation) methods. However, the improper use of these mechanisms leads to a range of memory-related errors that may result in unexpected program crashes and security vulnerabilities.

Memory safety is a property of programs that guarantees that memory accesses are always to defined and valid memory. Namely, memory objects are only accessed between their intended bounds, during their lifetime, and given their original or compatible type.

Languages with automatic memory management (e.g., with garbage collection) such as Java, abstract away memory management details and ensure memory safety in the implementation. On the contrary, the programmer and the compiler are responsible for handling this property in C and C++.

Taking into account the above definition, memory safety violations comprise any access to memory not defined by the program (i.e., at the time of dereference), and pointer dereferences to not designated memory regions. There are certain situations that infringe memory safety and lead to memory corruption. They are usually grouped into spatial errors, temporal errors, and type confusion vulnerabilities (which eventually lead to temporal or spatial safety violations).

```
char buffer[10];
char input[15];

strncpy(buffer, input, sizeof(
    input));
```

(a) Example buffer overflow

```
char *p = (char *)malloc(5*
    sizeof(char));

free(p);
p[0] = 'a';
```

(b) Example use-after-free

```
class Parent { int a; };
class Child: public Parent { int b; };

Parent *p = new Parent();
Child *c = static_cast<Child *>(p);
c->b = 0;
```

(c) Example type confusion

Fig. 1: Memory errors code examples

2.1 Spatial Errors

Spatial memory safety violations take place when a pointer is used to access a location in memory that is outside the bounds of the intended allocated memory object, resulting in accesses to not designated memory areas (i.e., to that pointer) or undefined memory.

Buffer overflows are the typical representative instance of this class of errors, when the program writes past the buffer boundary. To illustrate, Figure 1a shows a simple buffer overflow example. The `strncpy` function is called to copy the characters from `input` to `buffer`. However, the `input` array is longer than `buffer` and will overwrite adjacent memory contents.

2.2 Temporal Errors

Temporal memory safety violations occur when the program uses a pointer to access a memory object out of its lifetime. For instance, when dereferencing a dangling pointer. Within this category, we can find use-after-free errors for accesses to deallocated memory objects, double-free errors, accesses to not initialized memory, and Null-pointer dereferences. Other more specific instances exist, such as use-after-return for accesses to local objects after a function has returned.

Figure 1b shows a simple use-after-free example. Pointer `p` points to a dynamically allocated object in memory that is later freed in the execution. Afterwards, `p` is dereferenced, which results in undefined behavior and a possible exploitation opportunity.

2.3 Type Confusion

In languages that support the object-oriented programming paradigm such as C++, it is essential to be able to convert one object type to another, known as type conversion. Type confusion (or bad-casting) takes place when an object

type is interpreted as another due to unsafe type casting, which results in the wrong reinterpretation of the underlying type representation.

Among the provided types of cast by C++, `static_cast` checks whether the conversion is valid at compile-time, and is commonly used in performance critical applications because of its efficiency. However, it does not verify the safety of the conversion at run-time.

Programs often cast an instance of a parent class to a derived class (i.e., downcasting), which is not safe if the parent lacks any of the fields or virtual functions of the derived class. When the program then uses any of these fields, it may use data as a regular field in one context and as a virtual function table pointer in another, enabling memory corruption. Figure 1c illustrates this situation.

3 Memory Corruption Attacks and Mitigations

Starting from the first discussion in 1972 on how buffer overflows overwrite memory and could be used to inject code, an arms race began in which both attacks and countermeasures have been evolving with memory errors as the central piece [55].

Attacks are primarily intended to alter the execution of programs to make them behave in unintended ways, and usually take control. To this end, one of the most straightforward methods is to modify the program code that resides in memory through a buffer overflow, known as code corruption [51]. Nevertheless, this approach is conveniently prevented by setting code pages as read-only, leveraging the support from modern processors.

Control-flow hijacking forms the basis for plenty of attacks, making programs deviate from their intended control-flow [40, 17]. For example, an attacker can exploit a buffer overflow to overwrite the return address stored in the stack for the corresponding stack frame. When the function returns, the execution will continue at the new given address. To execute the code specified by the attacker, there are two strategies: code injection and code reuse. It is worth to mention that stack canaries are a widely used mechanism to prevent the corruption of return addresses in the stack. They insert crafted values around return addresses to later verify if they have been modified. However, attackers can leverage other code pointers to carry out attacks [51].

3.1 Code Injection

Within control-flow hijack attacks, code injection relies on injecting malicious code into the memory of the program (e.g., writing to a buffer) and using the previously described technique to direct the control-flow to it. This code traditionally spawned a shell.

To protect systems against this type of attacks, the Write XOR Execute protection policy enforces that memory pages can be writable or executable but not both at the same time, effectively removing the executable permission from data pages.

3.2 Code Reuse

In order to circumvent the Write XOR Execute policy that renders code injection attacks ineffective, new attacks reuse the own code of the program that is loaded in memory, chaining the execution of existing code pieces through code pointers [39].

The earliest implementation of this class of attacks is return-into-libc, which exploits memory corruption to make the program deviate from its control-flow and execute functions from the C standard library. By means of chaining the execution of libc functions, attackers are able to effectively take control [14].

Rather than reusing complete libc functions, Return Oriented Programming (ROP) generalizes code reuse attacks by targeting gadgets — interesting code sequences that end in a return instruction [44, 54]. When the execution of these gadgets is chained, the program is able to execute high-level tasks. On the basis of ROP, other code reuse variants have been developed, such as Jump Oriented Programming (JOP) [3], which targets gadgets that end in indirect branches instead of return instructions, or Block Oriented Programming (BOP) [21], that makes use of basic blocks as gadgets.

3.3 Control-Flow Integrity

To hinder control-flow hijacking, proposed approaches seek to preserve the integrity of the possible paths that programs may follow. The Control-Flow Integrity (CFI) policy relies on the construction of a Control-Flow Graph (CFG) of the program that represents the legitimate execution paths, and prevents any deviations from it during execution [5]. This is usually achieved through the instrumentation of indirect branches.

CFI implementations are often differentiated by the kind of policy they enforce. Forward-edge CFI is focused on jump instructions and indirect calls, whereas backward-edge CFI preserves the integrity of control-flow transfers from return instructions. Further, fine-grained CFI enforces a more strict policy [38], whereas coarse-grained CFI is less restrictive for better performance [53]. Unfortunately, more sophisticated code reuse attacks are able to bypass many existing implementations of CFI, taking advantage of object semantics in C++ programs [40], for instance.

3.4 Randomization-based Defenses

As an additional line of defense, randomization-based approaches make exploitation harder through probabilistic techniques. Since attacks need to know the specific addresses of their targets, randomizing the location of regions in memory renders them ineffective. Address Space Layout Randomization (ASLR) is a widely deployed method that randomizes the process memory layout by randomly placing memory regions such as program code, the stack, heap, and shared libraries.

However, attackers successfully identified and exploited weaknesses on these approaches. First, address randomization in 32-bit systems is susceptible to brute-force attacks due to low entropy [45]. In addition, taking advantage of memory corruption, attackers can leak memory contents to uncover the memory layout of the program [47].

In light of these issues, researchers enhanced existing randomization techniques to not be vulnerable to memory disclosure attacks. In that context, newer approaches use a shadow stack to protect code pointers [30]. Alternative approaches rely on systematic re-randomization of the address space to invalidate current code pointers [10]. However, attacks evolve at the same time, successfully bypassing modern randomization techniques by directly reusing the randomized code pointers [17].

3.5 Non-control Data Attacks and Mitigations

Different from the aforementioned attacks, non-control data attacks do not seek to hijack the control-flow of the program to carry out malicious operations. Consequently, existing mitigations that are focused on preventing alterations to the intended control-flow of the program are not effective against this class of attacks.

The ultimate goal of data-only attacks is to modify the data-flow of the program by altering security-critical data in memory, such as configuration data, decision-making values, and user identities, for instance. As a result, attackers are able to escalate privileges, circumvent authentication, or leak sensitive information [9].

`FlowStitch` [19] is a practical approach that automates the creation of data-only attacks through the exploitation of memory errors to maliciously join existing data-flow paths to leak or alter relevant data in the program. Data Oriented Programming (DOP) [20] generalizes non-control data attacks by systematically building exploits in a Turing-complete fashion using techniques to find gadgets and to chain them together in a conceptually similar vein to ROP, except that it does not modify any code pointers.

Equivalent to CFI for control-flow hijack attacks, Data-Flow Integrity (DFI) seeks to protect against data-only attacks by enforcing that data only follows legitimate data-flows. To this end, DFI uses reaching definitions data-flow analysis to construct a Data-Flow Graph (DFG) that maps the instructions to written values and uses of these values. By instrumenting read and write instructions, DFI verifies that data-flows are within the DFG at run-time [7, 48].

3.6 Additional Mitigations

Several alternatives exist to the previously mentioned mitigation strategies. For instance, pointer integrity policies are an efficient approach to protect program pointers. This method is used to protect code pointers against control-flow hijacking [26] and even data pointers for a more comprehensive protection [28].

Software Fault Isolation (SFI) is another method that isolates untrusted modules in a sandbox in the address space of the host program, in such a way that it cannot perform memory accesses out of the sandbox. To illustrate, Google NaCl [41] loads untrusted code into a specific memory area, and confines its memory and instruction references to the sandbox through instrumentation. Alternatievly, ARMlock [58] leverages ARM hardware support to set up sandboxes by assigning memory to different domains, and locking the execution to the current domain.

4 Memory Corruption Finding Techniques

Memory safety issues are of great relevance to computer systems. This is also evidenced by the large amount and diversity of approaches proposed to find them in the research community [51, 50]. These approaches seek to identify security vulnerabilities in software before it is finally deployed, and are based on static program analysis, dynamic program analysis, or a combination of both.

Finding bugs in operating system kernels presents unique challenges compared to user-level applications. This is attributed to their extensive and heterogeneous codebases, interaction with devices, privilege levels, and so on. Nevertheless, several approaches have been developed that specifically target kernel errors by building a *soundy* static analysis technique for taint and pointer analysis [31], or a dynamic analysis framework to hook and fuzz kernel drivers [49], for instance.

During the rest of this section, we mainly discuss the most representative general-purpose memory safety enforcement and error identification approaches not tied to specific software.

4.1 Fuzz Testing

Among the dynamic analysis techniques, fuzzing has been certainly one of the most active research topics in recent years [32]. From a broad perspective, the core of the iterative process of fuzzing has three different phases. First, it generates an input, or a set of inputs (i.e., test cases), to be fed to the program under test. Then, it executes the program given the generated inputs. Last, the observations produced from the execution are evaluated to determine whether the given input is interesting or not, so that the input generation subsystem will use it for further executions or not. Moreover, if the program crashes, it generates a crash report with relevant information to analyze and reproduce it. Each of the stages can be further divided into more specific tasks, we encourage the reader to refer to the literature for more in-depth information [24, 32]. Existing fuzzing techniques are also usually classified into black-box, grey-box, and white-box approaches depending on the information they observe and use during the process.

Dynamic analysis approaches are only able to detect vulnerabilities in a concrete execution, meaning that they only explore a small subset of the possible

program paths. Considering that fuzzing usually aims at exploring as many paths as possible and uses code-coverage metrics, combining it with sanitizers greatly boosts the ability to find memory errors [15, 36]. For the specific concurrency memory corruption vulnerabilities, current fuzzers have also successfully incorporated instrumentation and feedback mechanisms specific to threads and scheduling [8].

4.2 Static Approaches

Static analysis vulnerability detection refers to the set of program analysis techniques that are devised to analyze program code (i.e., in either high-level language, intermediate representation, or binary form), and reason about different program properties to check memory safety, producing results that are conservatively correct for every possible execution. Although static analyses are able to provide more complete results by taking into account more execution paths than dynamic analysis, they suffer from a higher false alarm rate, and usually need to make tradeoffs among soundness, precision and scalability.

Among the static approaches developed to detect memory corruption (i.e., each taking into account the different semantics of the different spatial and temporal errors), the most notable used techniques are: (i) abstract interpretation [11], which approximates the behavior of program execution and enables the computation of invariants; (ii) model checking [25], that computes and reasons about all program states; (iii) pointer analysis [57], which seeks to approximately determine the possible values of a pointer; and (iv) pattern matching [37], that reasons and finds occurrences of the given patterns.

4.3 Symbolic Execution

In a symbolic execution, programs are executed symbolically. That is, input values are represented as symbolic values instead of concrete values and the execution is carried out by a symbolic execution engine. These values are used to generate path conditions, which are logical expressions that represent the program state and the transformations between them. To implement symbolic execution, it is necessary to maintain a symbolic domain, which is composed of a symbolic state that maps variables to symbolic expressions, and a path condition formula for each program path. Conditional statements update the path condition formula, whereas assignments update the symbolic state. Several approaches exist within symbolic execution, such as forward and backward execution, or concolic execution [2].

Generally, symbolic execution is used to verify properties of programs (e.g., memory safety) and to automatically generate test cases to augment other dynamic analyses, for instance. In conjuntion with vulnerability models, researchers have successfully used symbolic execution to find memory errors such as heap overflows [23] and to also generate memory corruption exploits [46].

4.4 Sanitizers

In contrast to static analysis, sanitizers are dynamic analysis tools that instrument programs, usually at compile-time or at binary-level by leveraging Dynamic Binary Instrumentation (DBI) frameworks, to insert the so-called inlined reference monitors and perform several checks at run-time. In this way, they analyze an actual program execution and produce accurate results with regards to that specific execution.

Over the years, a great number of sanitizers have been proposed in order to identify and diagnose memory safety violations, each implementing methods to find spatial memory corruption, temporal memory corruption, or both. Typically, these approaches create and maintain metadata that describes the spatial or temporal properties respectively of a pointer or object.

In relation to spatial memory safety, we can mostly discern two kinds of approaches: those based on whitelisting memory regions, and those based on blacklisting. At the same time, existing approaches either maintain metadata for each pointer, or for each memory object. For instance, whitelisting and per-object bounds tracking approaches [1] store bounds metadata for each object in a bounds table, and check it on pointer arithmetic. Alternatively, per-pointer bounds tracking [33] stores bounds metadata for each pointer in a bounds table, propagates that metadata on pointer arithmetic, and checks it on each dereference. In contrast, blacklisting techniques [42, 43, 4] insert poisoned red-zones around memory objects, track the state of the program using a shadow memory, and check the shadow state on each memory access.

With regards to temporal memory safety, the main used techniques are lock-and-key and memory quarantining. Lock-and-key approaches assign a unique key to memory objects and store that value in a lock location in memory, keeping that metadata for each pointer, and checking on each pointer dereference [34]. When objects are deallocated the corresponding key is no longer valid. Instead, quarantine-based approaches put memory regions into quarantine as soon as they are deallocated, inserting red-zones, and delaying their possible allocation [42, 35] while checking memory accesses.

In order to detect type confusion, sanitizers add run-time checks to detect type casts that are incompatible. Existing tools either use run-time type information [52] or track type metadata for each object [18, 22] to verify `static_cast` operations, or check type compatibility on each pointer dereference [16].

We encourage the reader to refer to existing literature to find further information on sanitization techniques and implementations [50].

5 Related Work

Memory errors have been a popular topic of research for more than three decades. Therefore, researchers have carried out multiple works that review, survey, or systematize the knowledge on the different areas within this field.

For instance, Szekeres et al. [51] present a comprehensive analysis of memory corruption in which they develop a general model for attacks and protection

policies. For programs in their binary form, `angr` [46] is an extensive framework that implements numerous techniques to identify and exploit memory safety violations. From a different perspective, Van der Veen et al. [55] elaborate on the historical evolution of memory corruption. They point out that memory errors are unlikely to lose significance in the near future, and that despite all the proposed mitigations, attackers still manage to exploit memory errors.

In the context of detection techniques, Song et al. [50] bring together the knowledge about sanitizing methods and implementations. They present a comprehensive taxonomy of existing tools, highlighting several fundamental characteristics. Baldoni et al. [2] survey the essentials of symbolic execution and the most representative techniques for testing and security applications, whereas Klees et al. [24] thoroughly evaluate existing fuzzing implementations and their experimental methodologies.

In relation to attack mitigation strategies, Larsen et al. [27] systematize the understanding on software diversity techniques and point out fundamental trade-offs in fully automated approaches.

6 Conclusion

Memory safety violations are a major source of vulnerabilities in current systems. Over the decades, researchers have devised numerous methods to find and patch memory errors, or to protect systems from their exploitation. At the same time, attacks that are built upon memory corruption have been constantly evolving to circumvent newer defenses.

In this paper, we have scrutinized the characteristics of memory errors and the different attacks that exploit them. We have also analyzed the main detection approaches, and techniques to mitigate subsequent attacks. Despite all the proposed approaches, memory corruption persists and is actively exploited, suggesting that memory safety is still a serious concern, and that further research is needed. We hope that this paper will serve as a baseline to assist researchers to contribute to the field.

Acknowledgments

This work is partially supported by the Basque Government under a pre-doctoral grant given to Oscar Llorente-Vazquez.

References

1. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: USENIX Security Symposium (2009)
2. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) (2018)

3. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: ACM Symposium on Information, Computer and Communications Security (2011)
4. Bruening, D., Zhao, Q.: Practical memory checking with dr. memory. In: International Symposium on Code Generation and Optimization (CGO) (2011)
5. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Computing Surveys (CSUR) (2017)
6. Cai, Y., Zhu, B., Meng, R., Yun, H., He, L., Su, P., Liang, B.: Detecting Concurrency Memory Corruption Vulnerabilities. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (2019)
7. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2006)
8. Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H., Liu, Y.: MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In: USENIX Security Symposium (2020)
9. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: USENIX Security Symposium (2005)
10. Chen, X., Bos, H., Giuffrida, C.: Codearmor: Virtualizing the code space to counter disclosure attacks. In: IEEE European Symposium on Security and Privacy (EuroS&P) (2017)
11. Clang static analyzer. `https://clang-analyzer.llvm.org/`
12. Cloosters, T., Rodler, M., Davi, L.: Teerex: Discovery and exploitation of memory corruption vulnerabilities in sgx enclaves. In: USENIX Security Symposium (2020)
13. Cwe - 2021 cwe top 25 most dangerous software weaknesses. `https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html`
14. Designer, S.: Return-to-libc attack. Bugtraq, Aug (1997)
15. Dinesh, S., Burow, N., Xu, D., Payer, M.: Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In: IEEE Symposium on Security and Privacy (2020)
16. Duck, G.J., Yap, R.H.: Effectivesan: type and memory error detection using dynamically typed c/c++. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2018)
17. Göktas, E., Kollenda, B., Koppe, P., Bosman, E., Portokalidis, G., Holz, T., Bos, H., Giuffrida, C.: Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In: IEEE European Symposium on Security and Privacy (EuroS&P) (2018)
18. Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., Van Der Kouwe, E.: Typesan: Practical type confusion detection. In: ACM SIGSAC Conference on Computer and Communications Security (2016)
19. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: USENIX Security Symposium (2015)
20. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: IEEE Symposium on Security and Privacy (2016)
21. Ispoglou, K.K., AlBassam, B., Jaeger, T., Payer, M.: Block oriented programming: Automating data-only attacks. In: ACM SIGSAC Conference on Computer and Communications Security (CCS) (2018)

22. Jeon, Y., Biswas, P., Carr, S., Lee, B., Payer, M.: Hextype: Efficient detection of type confusion errors for c++. In: ACM SIGSAC Conference on Computer and Communications Security (2017)
23. Jia, X., Zhang, C., Su, P., Yang, Y., Huang, H., Feng, D.: Towards efficient heap overflow discovery. In: USENIX Security Symposium (2017)
24. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: ACM SIGSAC Conference on Computer and Communications Security (2018)
25. Kroening, D., Tautschnig, M.: Cbmc–c bounded model checker. In: Conference on Tools and Algorithms for the Construction and Analysis of Systems (2014)
26. Kuznetzov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: USENIX Security Symposium (2018)
27. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: Sok: Automated software diversity. In: IEEE Symposium on Security and Privacy (2014)
28. Liljestrand, H., Nyman, T., Wang, K., Perez, C.C., Ekberg, J.E., Asokan, N.: {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In: USENIX Security Symposium (2019)
29. Liu, C., Zou, D., Luo, P., Zhu, B.B., Jin, H.: A Heuristic Framework to Detect Concurrency Vulnerabilities. In: Annual Computer Security Applications Conference (ACSAC) (2018)
30. Lu, K., Song, C., Lee, B., Chung, S.P., Kim, T., Lee, W.: Aslr-guard: Stopping address space leakage for code reuse attacks. In: ACM SIGSAC Conference on Computer and Communications Security (CCS) (2015)
31. Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: DR.CHECKER: A soundy analysis for linux kernel drivers. In: USENIX Security Symposium (2017)
32. Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering (2019)
33. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Softbound: Highly compatible and complete spatial memory safety for c. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2009)
34. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Cets: compiler enforced temporal safety for c. In: International Symposium on Memory Management (2010)
35. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices (2007)
36. Österlund, S., Razavi, K., Bos, H., Giuffrida, C.: Parmesan: Sanitizer-guided greybox fuzzing. In: USENIX Security Symposium (2020)
37. Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in linux device drivers. Acm operating systems review (2008)
38. Payer, M., Barresi, A., Gross, T.R.: Fine-grained control-flow integrity through binary hardening. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (2015)
39. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC) (2012)
40. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In: IEEE Symposium on Security and Privacy (2015)
41. Sehr, D., Muth, R., Biffle, C.L., Khimenko, V., Pasko, E., Yee, B., Schimpf, K., Chen, B.: Adapting software fault isolation to contemporary cpu architectures. In: USENIX Security Symposium (2010)

42. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: USENIX Annual Technical Conference (2012)
43. Seward, J., Nethercote, N.: Using valgrind to detect undefined value errors with bit-precision. In: USENIX Annual Technical Conference (2005)
44. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: ACM conference on Computer and Communications Security (2007)
45. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: ACM conference on Computer and communications security (CCS) (2004)
46. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (2016)
47. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: IEEE Symposium on Security and Privacy (2013)
48. Song, C., Lee, B., Lu, K., Harris, W., Kim, T., Lee, W.: Enforcing kernel security invariants with data flow integrity. In: Annual Network and Distributed System Security Symposium (NDSS) (2016)
49. Song, D., Hetzelt, F., Das, D., Spensky, C., Na, Y., Volckaert, S., Vigna, G., Kruegel, C., Seifert, J.P., Franz, M.: Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In: Annual Network and Distributed System Security Symposium (NDSS) (2019)
50. Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: Sok: sanitizing for security. In: IEEE Symposium on Security and Privacy (2019)
51. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: IEEE Symposium on Security and Privacy (2013)
52. Undefinedbehaviorsanitizer. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
53. Van Der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A tough call: Mitigating advanced code-reuse attacks at the binary level. In: IEEE Symposium on Security and Privacy (2016)
54. van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., Giuffrdia, C.: The dynamics of innocent flesh on the bone: Code reuse ten years later. In: ACM SIGSAC Conference on Computer and Communications Security (2017)
55. Van der Veen, V., Cavallaro, L., Bos, H., et al.: Memory errors: The past, the present, and the future. In: International Conference on Recent Advances in Intrusion Detection (RAID) (2012)
56. Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., Sui, Y.: Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: IEEE/ACM International Conference on Software Engineering (ICSE) (2020)
57. Yan, H., Sui, Y., Chen, S., Xue, J.: Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: IEEE/ACM International Conference on Software Engineering (ICSE) (2018)
58. Zhou, Y., Wang, X., Chen, Y., Wang, Z.: Armlock: Hardware-based fault isolation for arm. In: ACM SIGSAC conference on Computer and Communications Security (CCS) (2014)