Automatically Assessing Crashes from Heap Overflows

Liang He*, Yan Cai†, Hong Hu‡, Purui Su*†§, Zhenkai Liang‡, Yi Yang*,
Huafeng Huang*, Jia Yan*, Xiangkun Jia*, Dengguo Feng*†

*Trusted Computing and Information Assurance Laboratory,
Institute of Software, Chinese Academy of Sciences, Beijing, China
†State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
‡Department of Computer Science, National University of Singapore
{heliang, purui, yangyi, yanjia, huafeng}@iscas.ac.cn, liangzk@comp.nus.edu.sg,
{ycai.mail, huhong789, ajiahit}@gmail.com, fengdg@263.net

***Abstract*—Heap overflow is one of the most widely exploited vulnerabilities, with a large number of heap overflow instances reported every year. It is important to decide whether a crash caused by heap overflow can be turned into an exploit. Efficient and effective assessment of exploitability of crashes facilitates to identify severe vulnerabilities and thus prioritize resources. In this paper, we propose the first metrics to assess heap overflow crashes based on both the attack aspect and the feasibility aspect. We further present** HCSIFTER**, a novel solution to automatically assess the exploitability of heap overflow instances under our metrics. Given a heap-based crash,** HCSIFTER **accurately detects heap overflows through dynamic execution without any source code or debugging information. Then it uses several novel methods to extract program execution information needed to quantify the severity of the heap overflow using our metrics. We have implemented a prototype** HCSIFTER **and applied it to assess nine programs with heap overflow vulnerabilities.** HCSIFTER **successfully reports that five heap overflow vulnerabilities are highly exploitable and two overflow vulnerabilities are unlikely exploitable. It also gave quantitatively assessments for other two programs. On average, it only takes about two minutes to assess one heap overflow crash. The evaluation result demonstrates both effectiveness and efficiency of** HCSIFTER**.**

***Index Terms*—Memory error, Heap overflow, Vulnerability assessment**

I. INTRODUCTION

Heap-based buffer overflow is one of the most widely exploited vulnerabilities in recent security incidents. High-risk heap overflow errors can be leveraged by attackers to execute arbitrary code or leak sensitive information (such as passwords and encryption keys) while low-risk heap overflow errors may only lead to Denial of Service attack (DoS). Given the large number of heap overflow errors reported every year, it is ideal to assess their severity efficiently and effectively, so that resources can be allocated towards high-risk ones for timely analysis and patching. The representative assessment of the severity is the exploitability – the likelihood that a heap overflow error can be utilized to launch the attacks to run arbitrary code.

An ultimate way to demonstrate the exploitability of a vulnerability is to generate working exploits. Previous works [1], [2], [3], [4] study the practical viability to automatically generate exploits. One mechanism is to symbolically execute the whole program and capture the program constraints as predicates. Then it generates working exploits by solving the constraints. AEG [1] and Mayhem [2] are representative ones in this category. Though using exploit generation as a way for assessment is accurate, they cannot generate exploits for all potential vulnerabilities due to the limitation in program analysis techniques.

Another assessment method is to analyze the code executed after the ***crash point*** – the instruction leading to the program crash. For example, the !exploitable tool [5] developed by Microsoft checks all instructions in the same basic block as the crash point to find ***exploit points*** – the exploitable special-purposed instructions, like control-flow-transfer instructions (e.g., `call` and `jmp`) or condition-affecting instructions (e.g., `cmp` followed by `jnz`). CRAX [3] takes a mixed method with these two techniques: starting from the crash point, it symbolically executes the program to find exploit points. The weakness of this type of solutions is the correctness of results. The corruption usually affects the original program behavior (e.g., due to crashes on dereferencing pointers from the overflowed memory), preventing these techniques from finding any exploitable points. Moreover, these tools will stop work when a crash occurs, leading missed potential exploit points.

In this paper, we aim to address the challenge in automatically assessing the exploitability of crashes caused by heap overflow when no working exploits can be easily generated. We propose a set of metrics to quantify the exploitability of heap overflows. To the best of our knowledge, this is the first approach that aims to automatically assess the exploitability of heap overflows. Our metrics are based on two aspects:

§Corresponding author.

the *Attack Metrics* and the *Feasibility Metrics*. The former measures the potential threats of a heap overflow and the latter measures the specific difficulties to build a real working exploit.

Based on the metrics, we present a framework, HCSIFTER[1], for automatic heap overflow crash assessment. HCSIFTER accurately detects heap overflow errors through dynamic program analysis. It tracks all heap objects and checks related operations to find the out-of-bound heap memory accesses. The detection happens right before the real execution of the ***corruption point*** – the instruction leading to the heap overflow. To extract features of the overflow's impact on the program execution, HCSIFTER dynamically carries out *data recovery* in the corrupted memory region, so that the program can continue its normal execution for our analysis. Note that, compared with other dynamic approaches, this is a key step to continue the execution after a heap overflow occurrence without any crashes, enabling HCSIFTER to explore additional exploitable points. During the dynamic subsequent execution, HCSIFTER tracks all recovery memory regions and detects exploits based on exploit patterns. We also identify several novel strategies that enable HCSIFTER to detect exploit points missed by other techniques.

In this paper, we aim to assess a single path from a given crash. If multiple paths exist, our technique can be applied on each path separately. It is possible to find other paths from a crash with fuzzing tools [6], [7] or symbolic execution tools [8], [9].

In summary, the contributions of this paper include:

- We propose a set of novel metrics for quantifying the severeness of heap overflow crashes. Our metrics measures the challenges to convert crashes into exploits, including both the potential attack aspect and the feasibility aspect.
- We design and implement a prototype tool HCSIFTER, which automatically assesses the exploitability of given heap overflows based on our metrics. HCSIFTER introduces dynamic memory recovery and second-order overflow to automatically assess the difficulties for exploit generation.
- We evaluated HCSIFTER using real-world vulnerable programs. The results demonstrated both the effectiveness and efficiency of HCSIFTER.

II. Problem Definition and Challenges

In this paper, we aim to assess the exploitability of given heap overflow errors. We define the problem as follows:

Assessment of Heap Overflow Crash: Given a binary with a heap overflow bug, and a proof of concept (PoC) that crashes the program, we aim to quantify the severeness of the error, i.e., the likelihood for attackers to further develop this crash into a working exploit, such as arbitrary memory access or code execution attacks.

[1]Standing for *Heap Crash Sifter*.

TABLE I
Exploit Point Types. "taint" means user input data.

Exploit Points	Description
call/jmp taint	Directly call/jump into the tainted target
call/jmp [taint]	Indirectly call/jump into the target with a tainted address
mov [taint1], taint2	Write a tainted value taint2 into the memory location with a tainted address taint1
mov [taint1], [taint2]	Read the value at a tainted address taint1 and write it to the tainted address taint2
critical_func(taint)	Control the arguments of security-critical functions

Different from exploit generation [1], [2], [3], [4], which generates a concrete working exploit, we aim to estimate the level of difficulty for attackers to build a working exploit. The output should be a quantitative assessment that describes the severity of the error. For example, program developers can use the assessment result to prioritize the fix of highly exploitable overflows, considering the limited human resources. Further, the assessment result can also be used by exploit generation tools to focus on overflows with high exploitability.

Challenges in Assessment: Our approach is based on dynamic program analysis to address the problems. Several challenges must be resolved:

- **C1: Preserving the program execution.** In the dynamic code analysis, the first consideration is to continue the program execution after the corruption point, which is required by our approach to further analyze the impact of the overflow. Directly continuing the execution after a heap overflow is infeasible, as the subsequent execution usually crashes due to invalid memory access.
- **C2: Bypassing the integrity checks.** It is problematic if a heap overflow overwrites the meta data of free heap chunks, such as the backward and forward pointers, which are protected by heap manager. If the overflow has to overwrite the meta data to reach one exploit point, the attack needs to bypass the integrity checks of heap data structure.
- **C3: Preserving the memory layout.** Heap memory is dynamically allocated by the heap manager. The order of heap allocation and free significantly affects the memory layout. Analysis code in the same memory space with the vulnerable program may change the memory layout, and thus affect the assessment of the exploitability. The analysis should be performed in a different memory space from the vulnerable program.

III. Exploitability Metrics

As one of our main contributions, we first propose two kinds of metrics to measure the severity of heap overflow crashes. The metrics should not only reflect the possible exploit methods an attacker may choose, but also the difficulty to build working exploit in real-world programs. Specifically, we define two types of metrics to assess the features of the crashing program: *Attack Metrics* and *Feasibility Metrics*.

A. Attack Metrics

Attack metrics describe the potential risks that a heap overflow may introduce. Intuitively, if a heap overflow introduces more risks (e.g., corrupt more function pointers), it becomes easier for attackers to utilize the overflow.

1) **Exploit Point (Exp)**: As different exploit points have different attack power, we define five types of exploit points as shown in Table I, where *taint1* and *taint2* indicates the user input data. This metric is crucial as more exploit points means higher exploitability of the error.
2) **Overflow Bytes (OB)**: A larger memory area corrupted by a heap overflow usually means a higher threat. And it also means that more exploit points may be found. We use the number of the overflowed bytes to depict the basic information of a heap overflow.
3) **Taint Bytes (TB)**: For attackers who exploit a heap overflow to execute arbitrary code usually need to fill a proper length of payload within the input data. It is a common fact that a more complex payload needs more space to be filled with. So having more taint bytes means the ability to fill more complicate payload.
4) **Taint Relation (TR)**: We need to confirm that the overflow bytes are tainted (i.e., from user input data). If so, attackers could provide arbitrary payload (i.e., malicious code) to fill with them. We use this metric as the basic information of heap overflow.

B. Feasibility Metrics

Feasibility metrics depict the difficulties to build a real working exploit. We define two kinds of feasibility metrics. The first one is the count of various pointers corrupted by heap overflow, and the second one is the value constraint of overflowed bytes. Intuitively, if a heap overflow involves more such elements (e.g., more pointers that will be dereferenced), it becomes more difficult to exploit the overflow.

1) **Pointer Dereference Count (PDC)** As one main challenge faced by attackers is to ensure that all the memory accesses along the program path during an attack are not affected. Otherwise, memory dereference may lead to the program crash due to corrupted pointers. Based on the types of pointers possibly corrupted, we define the pointer dereference count as a four-tuple which includes the counts of four types of pointers:
   - **Index Pointer (IP)** where the corrupted data is an index pointer (offset) of a memory access, for example, `mov eax, [ebx + esi × 5]`. The register `esi` is used as the index pointer in the memory access and a memory access error will happen if `esi` is loaded with a large number. However, it is not difficult for attackers to preserve the correctness of such memory accesses (e.g., set the register `esi` to be `0`).
   - **Base Pointer (BP)** where the corrupted data is a base of a memory access for fetching data, for example, `mov eax, [ebx + esi × 5]`. The register `ebx` is used as the base pointer for memory access. Due to the address randomization [10], it is challenging to guess all the randomized pointer values correctly to avoid the crash.
   - **Multilevel Pointer (MP)** where the corrupted data is an pointer to another pointer. Compared to BP, where attackers can search and use any existed memory, it is more challenging for attackers to find a proper pointer's pointer.
   - **Protected Pointer (PP)**: OS or special program logic checks or protects whether a pointer is corrupted, such as the Safe Unlinking of Windows. This usually means that the exploiting process is very difficult and attackers have to bypass the protection checking.

   Specifically, based on the real-world situation, we define the order of difficulty levels for these metrics as follows, with increasing difficulty to exploit:

   $$\textbf{IP} < \textbf{BP} < \textbf{MP} < \textbf{PP}$$

2) **Value Constraint (VC)**: Different applications usually have their own strategies to check the validation of various input data. In this work, we only extract the value constraints for the data used to overwrite heap memory. Note that, VC does not directly affect our assessment; however, it plays an important role in building working exploits as it strictly indicates that some characters (e.g., 0x0, 0xA) cannot appear in the designed payload by attackers.

IV. HCSifter Framework

We present the design details of our framework, HCSifter. As shown in Figure 1, HCSifter works in three steps: heap overflow identification, exploit point shortlisting, and exploitability assessment. It tracks all heap-related operations with a multi-semantic taint analysis engine, which treats the input PoC, heap object, and recovery memory as different kinds of taint data. Then HCSifter detects cross-boundary heap accesses at both the instruction level and the function level (Section IV-A). It then identifies memory objects that can be exploited by attackers (Section IV-B). We further analyze exploitable code patterns and identify second-order overflow attack techniques. In the last step, for each exploitable code pattern, HCSifter analyzes related memory operations to evaluate the difficulty level of exploitability (Section IV-C).

A. Heap Overflow Identification

The goal of this step is to identify the location of the heap overflow vulnerability, so that we can record and recover the corrupted data by heap overflow. HCSifter utilizes dynamic taint analysis [11] to identify heap overflows. We firstly taint the base address and the size of the allocated heap blocks. When accessing tainted memory addresses, we check whether the access is out of the bound of the allocated heap blocks.

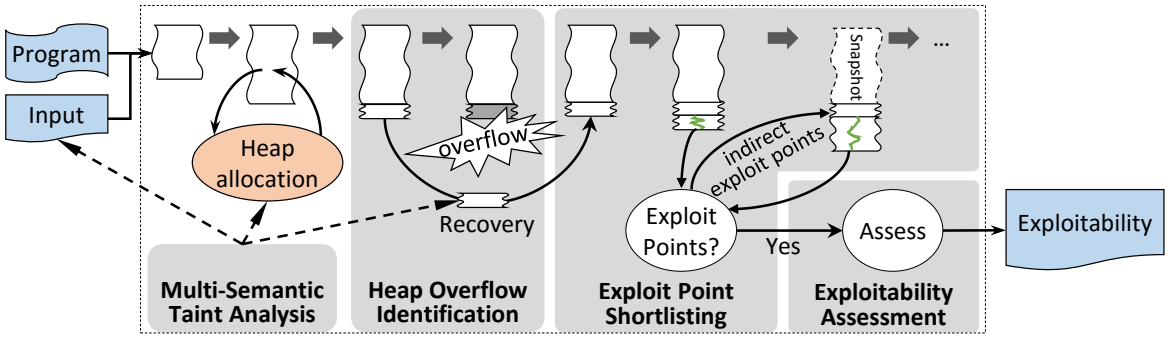

Fig. 1. The HCSIFTER framework. HCSIFTER takes as inputs the binary of the vulnerable program and the error-exhibiting input. It executes the program in a virtual environment to detect the heap overflow at run-time. Then HCSIFTER restores the memory to continue the execution. At last, it detects the exploit points and evaluates the exploitability.

Taint Source. To label information of the allocated heap blocks as tainted, HCSIFTER hooks the heap allocation functions (e.g.,`RtlAllocateHeap` on Windows system).

Taint Propagation. HCSIFTER propagates the taint value at the instruction level using the standard taint propagation mechanism. Due to taint explosion [12], HCSIFTER does not take control-flow dependencies into account. More details on taint analysis can be found in [13], [14].

Taint Checking. With the base address and the size information propagated by taint analysis, HCSIFTER checks every memory access to identify heap overflows. Specifically, for an instruction involving a pointer, it retrieves the taint value of the pointer operand used for the memory access. From the taint value, it restores the base address of the heap object and the allocated size. It then checks whether the pointer value falls into the base address and the end address of the heap object.

B. Exploit Point Shortlisting

Once the heap overflow vulnerability is detected, HCSIFTER analyzes the heap memory that can be corrupted by the overflow to identify data types (discussed in the second point below) that can be exploited by attackers. We first present the techniques for recovering the program's execution after the heap overflow.

1) Dynamic Heap Data Recovery: As discussed in Section II, the first challenge to detect exploit points is to preserve the program execution after the memory corruption. To address this challenge (**C1**), HCSIFTER recovers the corrupted data back to the uncorrupted version, so that the program can continue its execution without any crash. By watching the following executed instructions, we can find the usage of the corruptible data. The usage of the data is important to assess the exploitability. We use dynamic taint analysis to track the corruptible data. During the dynamic data recovery, we label each recovered data with the offset from the first overflowed bytes. In the following execution, we propagate the taint value as for the heap pointers.

2) Exploit Point Detection: We aim to detect the exploit points – the instructions that can be used by attackers to launch attacks – based on the data type affected by the heap overflow. We classify data into types like code pointers, data pointers, loop bounds. The data type can be inferred by the program's behavior. A heap data can *directly* affect program's code execution or the memory access if some program variables derived from it (tainted by its offset) are used as pointers to code and data. In addition, a heap object can *indirectly* affect memory accesses if memory addresses are *control-dependent* on it. Control-dependent means the heap object affects the control flow, and at the same time, the memory access happens inside the code under the control. For example, if a heap data is used as a counter of one loop, the memory access in the loop body is control-dependent on this heap data.

3) Exploitability by Second-Order Overflow: To address the challenge (**C2**), we identify an interesting heap meta data that can cause second-order heap overflows. It is the `size` field in the header of a free chunk. The method to trigger the second-order overflow is to change the value of `size` field into a larger value. We call this method *SizeOverflow*.

The basic idea is, during heap allocation, if the heap manager checks the value of `size` field in the header of the overflowed free chunk, whose value has been tampered be to a larger one (e.g., `0xFFFF`), then this free chunk will be allocated. The following memory access into this chunk may lead to new buffer overflows, as its size is believed to be larger than the actual one. This opens a new chance to explore additional exploit points. Compared with the exploit type defined in Section III-A which are Direct Exploit Point **(DExp)**, we call this data as Indirect Exploit Point **(IExp)**.

C. Exploitability Assessment

Besides the exploit points, HCSIFTER collects the feasibility metrics, especially the count of pointer dereferences, from the corruption points to exploit points. This kind of metrics will depict the real difficulty to reach the exploit points. We assess the exploitability with two basic rules as follows:

- **Rule-1:** If there is no pointer dereference from the corruption point to any direct exploit point, we treat it as an EXPLOITABLE error. If there is any pointer dereference from the corruption point to any exploit point, we treat it as a DIFFICULT error and we use nearest exploit point's highest difficulty level, defined in Section III-B as the assessment result.

TABLE II

Exploitability Assessment Overview. The 2nd main columns give the details of heap overflow. (1-1)* means indirect table look-up operation, B!0x0 means each byte cannot be 0x0, B!0xA means each byte cannot be 0xA. []+ means further exploitability is possible with a higher-order (> 2nd-order) overflow.

Programs	Heap Overflow Information			Basic Metrics			Exploitability Assessment	
	Instruction	Function	OB	TB	TR	VC	!exploitable	HCSIFTER
1ClickUnzip	mov [edx],al	lstrcpyA	2433	6573	(1-1)	B!0x0	✓	[IP]+
Acousitca Converter	mov [eax+edx], cl	lstrcpyA	1092	10613	(1-1)	B!0x0	UNKNOWN	[BP+PP]
CoreFTP Client	rep movsd	recv	8910	17653	(1-1)	B!0xA	UNKNOWN	[IP]
FoxitReader	mov [edx],al	lstrcpyA	4241	443	(1-1)*	B!0x0	✓	✓
HTTPDX	rep movsd	memcpy	1049	1059	(1-1)	B!0xA	UNKNOWN	[BP+PP]
Python	rep movsd	memcpy	64745	723419	(1-1)*	B!0x0	✓	✓
Vallen Zipper	mov [edx],al	lstrcpyA	4021	8443	(1-1)	B!0x0	UNKNOWN	✓
WMPlayer	rep movsd	-	2524	2547	(1-1)	-	✓	✓
ZipItFast	-	ReadFile	4021	20899	(1-1)	-	✓	✓

- **Rule-2:** If the error is reported as DIFFICULT according to Rule-1 and there exists any indirect exploit point from corruption point to the nearest exploit point, we enable the second-order overflow and repeat the assessment with the rule above.

We measure the pointer dereference as follows:

- We collect all instructions between the corruption point and the exploit point that access tainted address to deference memories, and put them into a set `S`.
- We retrieve the offsets of the bytes that affect the exploit point, and find the maximum offset `Max`.
- For any instruction inside `S`, if its memory operand is tainted with offset `off` and `off` $\leq$ `Max`, we count the dereferences by its types.
- The final value of the `derefCount` is the number of necessary tainted dereferences.

V. Implementation and Evaluation Summary

A. Implementation

We extend QEMU [15] to support the instruction instrumentation and the dynamic taint analysis outside the guest system. Unlike the TEMU [16] that needs to install extra drivers into guest system, we only use the emulated hardware information provided by QEMU to implement our system. By doing so, we can preserve the memory layout of guest system(**C3**).

HCSIFTER uses the udis86 library [17] to help disassemble x86 instructions. Our implementation supports about 200 x86 instructions for dynamic taint analysis, including the special support for float point registers (FPU) and SSE registers (XMM, MMX). Our implementation totally contains more than 36,000 lines of code, including 12,000 lines of C code for dynamic instruction instrumentation and data recovery in QEMU and 24,000 lines of C/C++ code for multi-semantic taint analysis and exploitability assessment.

B. Summary of Evaluation

1) Benchmarks and Experiment Setup: We configure HCSIFTER to run on a platform with 8 core-CPU, 8GB RAM, installed with Ubuntu 16.04 (x86-64) system. We run Windows XP-SP2 as a guest virtual machine to execute the vulnerable programs. To evaluate HCSIFTER, we collect 9 Windows programs with heap overflows from exploit-db [18]. All these programs are available either on the exploit-db site or at their official websites.

2) Efficacy in Exploitability Assessment: we apply HCSIFTER on the nine real-world heap overflow programs with crash PoCs and compare the result with the widely used tool !exploitable. HCSIFTER demonstrates its ability to accurately locate heap overflow instructions, which is the base of our assessment. As shown in the column of `Heap Overflow Information`, HCSIFTER confirms all nine known heap overflows. The major column `Basic Metrics` in Table II shows the assessment reported by HCSIFTER. In the column `Exploitability Assement` of Table II, we show the exploitability assessment for all nine programs. It reports that five programs can be "directly exploited" and two programs almost cannot be exploited, unless the attackers have the ability to bypass the integrity checker (`[BP+PP]`). Besides there are two "easily exploited" (1ClickUnzip and CoreFtp Client) as the attack is constrained only by index pointers (`[IP]`).

3) Importance of Indirect Exploit Points and SizeOverflow: Indirect exploit points play an important role in the exploitability assessment. Among five exploitable programs, one of them (i.e., FoxitReader) can only be easily attacked with the indirect exploit points, as it has one `BP` before the direct exploit point in the first round. Two programs, the 1ClickUnzip and the CoreFTP Client, only leave the exploit chance in the indirect exploit points. For 1ClickUnzip, the indirect exploit points in the second round provides further exploitability.

4) Assessment Performance: Our performance evaluation shows that HCSIFTER can finish one assessment efficiently, with less than 2 minutes on average and at most 5 minutes. For the memory overhead, HCSIFTER uses 52 MB more memory on average than the original QEMU execution. Such a memory consumption is acceptable on modern systems.

VI. Related Work

There are two main aspects related to our work in this paper. The first one is the automatic exploitation generation. The

second is the detection of software overflows, especially the heap overflow as focused in this paper.

Automatic Exploitation Generation. Brumley *et al.* [19] proposed the first patch-based automated exploitation of software vulnerabilities. Later, by integrating preconditioned symbolic execution and dynamic instruction instrumentation, Avgerinos *et al.* [1] implemented the first end-to-end system for fully automatic exploit generation. In practice, it is common that program source code is unavailable. Therefore, binary code based exploit generation is required. Mayhem [2] is the first one practically targeting on binary programs to automatically generate exploitation. PolyAEG [20] further targets to generate multiple exploits for a given vulnerable program based on control flow hijacking and redirection. FlowStitch [4] targets to automatically generate data-oriented exploits by searching ways to join program data flows.

Detection of Heap Overflows. Robertson *et al.* proposed to append additional protection data at the head or the tail of a heap to detect buffer overflows [21]. During a buffer overflow, the protection data is broken and can then be detected. Besides the protect data, inaccessible memory pages could also be allocated to detect buffer overflows [22]. Low-fat pointer [23] takes a method to detect heap overflow at run-time. It encodes the heap object information in addresses, to propagate the information, and to detect the buffer overflow.

The most related work to our approach is proposed by Slowinska *et al.* [24], which is based on binary data structure reversal. The approach assigns different colors to different heaps and monitors each heap access. However, this approach heavily relies on binary data reversal, which will result in imprecision (i.e., false negatives).

VII. Conclusion

Heap overflow is a severe threat to computer programs. However, it is a tedious work to determine whether a heap overflow is exploitable or not. In this paper, we propose HCSifter, a platform that automatically evaluates the exploitability of given heap overflows. HCSifter incorporates two kinds of metrics proposed in this paper to assess a heap overflow crash. We implement HCSifter as a prototype and evaluate it with nine real-world vulnerable programs. The experimental results show that HCSifter is effective and efficient on evaluating heap overflows.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61602457, 61572483, 61502469, 61502465), National 973 Program of China (2014CB340702) and the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (2017151).

References

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[2] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[3] S. K. Huang, M. H. Huang, P. Y. Huang, and C. W. Lai, "CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations," in *Proceedings of the 6th International Conference on Software Security and Reliability*, 2012.

[4] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic Generation of Data-Oriented Exploits," in *Proceedings of the 24th USENIX Security Symposium*, 2015.

[5] "!exploitable Crash Analyzer," http://msecdbg.codeplex.com/.

[6] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[7] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, January 2012.

[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[9] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[10] PaX Team, "PaX Address Space Layout Randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[11] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," in *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[12] E. Andrey, K. Sachin, S. Shenker, L. Fowler, and M. McCauley, "Towards Practical Taint Tracking," in *Technical Report No. UCB/EECS-2010-92*, 2010.

[13] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[14] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards efficient heap overflow discovery," in *26th USENIX Security Symposium*, 2017.

[15] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the USENIX Annual Technical Conference*, 2005.

[16] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[17] "Udis86 Disassembler Library for x86 and x86-64," https://github.com/vmt/udis86.

[18] "Offensive Security Exploit Database Archive ," https://www.exploit-db.com/.

[19] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proceedings of the 29st IEEE Symposium on Security and Privacy*, 2008.

[20] M. Wang, P. Su, Q. Li, L. Ying, Y. Yang, and D. Feng, "Automatic Polymorphic Exploit Generation for Software Vulnerabilities," in *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks*, 2013.

[21] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur, "Run-time Detection of Heap-based Overflows," in *Proceedings of the 17th USENIX Conference on System Administration*, 2003.

[22] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis, "A Dynamic Mechanism for Recovering from Buffer Overflow Attacks," in *Proceedings of the 8th International Conference on Information Security*, 2005.

[23] G. J. Duck and R. H. C. Yap, "Heap Bounds Protection with Low Fat Pointers," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[24] A. Slowinska, T. Stancescu, and H. Bos, "Body Armor for Binaries: preventing buffer overflows without recompilation," in *Proceedings of the Usenix Technical Conference*, 2012.