WinHeap Explorer: Efficient and Transparent Heap-based Bug Detection in Machine Code

Maksim Shudrak
Cloud Security and Quality Technologies Department
IBM Research Israel
Haifa, Israel
Email: maksims@il.ibm.com

***Abstract*—Despite all the efforts of the research community, buffer overflows remain one of the most dangerous bugs for modern IT systems. The problem is compounded by the fact that there are many developers who do not follow the basic rules of a secure software development lifecycle, supplying proprietary vulnerable products. To address this problem, the industry has proposed a number of techniques that perform analysis at the binary level. While most of them focus on problems that lead a program to crash or exception, there is a large class of more complex bugs that it is not possible to detect using only this criterion.**

In this paper we propose WinHeap Explorer, a high-performance solution for heap based bug detection in machine code using an original approach called light-weight dynamic binary instrumentation. The light-weight instrumentation is based on preliminary static analysis of code paths to highlight potentially erroneous parts, due to which we are able to decrease the overhead. Moreover, WinHeap Explorer does not change any memory allocation mechanisms, which preserves transparency of the tool towards the operating system. Our experiments have proven the ability of WinHeap Explorer to detect heap-based bugs as well as decrease runtime overhead for widely-used complex applications ranging between 24.2%-71.1% along with the same level of memory overhead in comparison with existing solutions. WinHeap Explorer is distributed under BSD license and available at [1].

***Index Terms*—Bugs, heap overflows, light-weight instrumentation, vulnerabilities, performance, overhead.**

I. Introduction

A modern software product is often a complex engineering system consisting of numerous sets of interrelated components that may have millions lines of code. The development of such complex software products is always accompanied by bugs that can pose a serious threat to the security of IT systems where vulnerable software is operated.

Buffer overflow remains one of the most dangerous and frequent types of vulnerabilities despite of all efforts undertaken to decrease and neutralize it during the last decade [2]. In this research, we focused on bugs that do not lead a program to crash. Thus we focus on bugs related to heap overflows and use after free, since stack overflows are more often accompanied with exceptions. According to the x86 architecture, a function's return address is stored on the stack, which significantly increases the probability of a crash or an exception in the case of overflow. Moreover, stack overflows are much better studied than heap overflows. For example, modern compilers offer a stack canary feature activated by default [3], [4], thereby reducing the risk that stack based buffer overflow will remain undetected. In addition, recent critical vulnerabilities in popular software, such as Symantec [5] and Adobe Flash [6], have shown that heap based overflows deserve the special attention of researchers.

```
1  IFileOpenDialog *pFileOpen;
2  void *a = malloc(some_mem_size);
3  // Create the FileOpenDialog object.
4  hr = CoCreateInstance(CLSID_FileOpenDialog, NULL,
       CLSCTX_ALL, IID_IFileOpenDialog, reinterpret_cast<void*
       *>(&pFileOpen));
5  // Show the Open dialog box.
6  r = pFileOpen->Show(NULL);
7  ...
8  // A part of the code is skipped
9  ...
10 memcpy(a, user_controls_this_input,
       user_controls_this_variable);
11
```

Listing 1: Heap overflow that may lead to vtable corruption

In our study, we decided to work only with x86 machine code due to the high presence of proprietary software that is distributed without source code for x86 architecture, as well as because this problem is less studied than analysis when source code exists. Since the vast majority of executables and libraries without available source code are supplied for Windows, we have limited our research only to this OS despite the fact that the problem also exists in Linux.

The term heap-based bug used in this paper includes the following list of bugs: heap overflows (HoF), heap underflows (HuF), heap overruns (HoR), heap underruns (HuR) as well as use after free bugs (UAF). We use term library call (libcall) when it comes to intermodule interaction (e.g. WinAPI call).

Let's consider the simplified motivational example in Listing 1. As outlined above, we consider x86 machine code while providing source code for better readability here.

At line 4, the WinAPI function `CoCreateInstance` creates a single object `IFileOpenDialog`, allocates memory in the heap and then saves pointers to accessible methods for this object in the allocated virtual table (vtable). As a result, `pFileOpen` is a pointer to the vtable. The code at line 6 contains a call to one of the accessible methods for `IFIleOpenDialog`. Then, the program allocates a block

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.20

of memory where input data is written in an amount which is controlled by user. Thus, we have the possibility of heap overflow at line 10. The bug potentially allows an attacker to execute arbitrary code in the context of a vulnerable application, in case he or she has the ability to rewrite the allocated vtable, for example, using a heap spray attack [7].

Detecting this type of bug is complicated, since even if we manage to rewrite the vtable during software testing, we still need the software under test (SUT) to call one of the methods from the rewritten vtable, leaving a high chance that the vulnerability will stay uncovered during analysis. It is important to note that vtable corruption is only one example, there are more complex heap-based bugs, for instance, when heap overflow leads to rewriting variables responsible for control-flow management.

There are a range of various solutions to detect such bugs in machine code based on static analysis [8], [9], [10], dynamic binary instrumentation (DBI) [11], [12], [13], [14], [15], [16], [17] and special debug allocators [9], [18], [19], [20], [21]. Unfortunately, all existing approaches have their own disadvantages:

- Static analysis is limited by the fundamental problem of binary code decompilation and allows detecting only basic types of bugs.
- DBI introduces serious performance overhead for the SUT, which significantly reduces the ability of its application for software testing.
- Special Debug Allocators may also introduce serious performance overheads for the SUT, and most of them change the standard mechanisms of memory allocation and often require source code access.

The main goal of our research is to develop a scalable and transparent tool for heap-based bug detection in the machine code of widely-used complex applications. We focused on implementing a tool that would be possible to integrate within existing black-box crash-oriented approaches such as fuzzing [22]. WinHeap Explorer has been implemented on the top of the Intel Pin DBI framework [23]. The tool performs bug detection by applying boundary checking for each instrumented memory access in the SUT, without any change to the standard mechanisms of memory allocation and does not require source code access.

The contributions of this paper are the following:

1) We propose an original approach called `light-weight instrumentation`. Under this approach, we combine the advantages of static and dynamic analysis, focusing instrumentation only on the code paths that are more prone to have bugs.
2) Based on the proposed approach, we implement an open-source tool called WinHeap Explorer which is available online at [1] and distributed under the BSD license.
3) Our experiments (Section VI) have confirmed the viability of light-weight instrumentation and have demonstrated its ability to decrease runtime overheads on average 24.2%-71.1% for widely-used complex applications in comparison with existing solutions as well as to detect heap-based vulnerabilities in widely-used complex applications.

The main idea of light-weight instrumentation is based on analysis of the most "interesting" code paths where the probability of error is higher. As such interesting code paths we select routines that perform memory manipulations or user input handling. This choice is dictated by our previous empirical investigations of bugs and vulnerabilities that appears in widely-used complex applications [24] as well as by other independent studies [25] [26]. It is sufficient to note that to get a list of these routines, preliminary instrumentation of the system and the SUT's dynamic-link libraries (DLLs) is performed (see Section V).

II. Related Works

A number of systems [8], [9], [10] perform static analysis of machine code, searching for vulnerable patterns using disassembling and decompilation. Unfortunately, correct decompilation of machine code is a theoretically unsolvable task, since compilation is a one-way process where important information in term of high level analysis is not included in the executable [27]. This makes it impossible to get an appropriate level of abstraction from machine code which is significantly reduce efficiency of static analysis. Moreover, modern compilers may introduce undefined behavior due to optimizations [28]. In general, the problem is called What You See Is Not What You eXecute (WYSINWYX) [29].

Another type of system installs special debug allocators or performs dynamic binary instrumentation to detect bugs at runtime.

Under the first approach, a standard memory allocator is replaced by a custom one. To detect a heap out of boundary access, the systems install special inaccessible memory zones (`page guards`) or specific values (`stack canaries`) around each allocated heap memory block, such as ElectricFence [20], GuardMalloc [18], PageGuard [19] and Dmalloc [21]. The significant disadvantage of such approaches is modification of system libraries and intervention in the standard process of memory allocation, which violates transparency of the testing tool towards the SUT. This is especially important for Windows where heap memory allocation is a complex undocumented process.

In WinHeap Explorer we also set up special zones (`red zones`) before and after each allocated heap block, writing them in a separate memory location called `shadow memory` (Section IV) which does not break transparency and allows the detection of incorrect access as soon as it happens in the SUT.

The second class of tools applies instrumentation to detect incorrect access to the heap. There are systems that use compile-time instrumentation such as AddressSanitizier [16], LBC [14], Purify [15], BoundLess [12], MudFlap [13] and runtime instrumentation such as Valgrind Memcheck [17] and DrMemory [11]. Unfortunately, compile-time instrumentation requires source code access, while runtime instrumentation introduces significant overhead.

DrMemory [11] is a memory checking tool implemented on top of the DynamoRIO [30] DBI framework. DrMemory performs instrumentation at runtime and wraps heap functions to install redzones before and after each allocated heap memory block which introduce significant runtime overhead for SUT as mentioned above.

AddressSanitizier [16] is a state of the art tool for memory bug detection. The system has been integrated in to the Chromium project [31] and is used with fuzzing to detect incorrect memory accesses during testing. Unfortunately, AddressSanitizier needs source code, which makes it impossible to detect bugs in closed source libraries and applications.

In general, the main difference between our solution and the above tools is that we use preliminary static analysis to highlight potentially erroneous parts of the code due to which we decrease runtime overheads. Also, WinHeap Explorer does not replace standard memory allocators, works at the binary level without any additional requirements for the testing environment and detects bugs with an accuracy of concrete machine instruction address and concrete byte in the memory. Moreover, WinHeap Explorer is thread safe and can be used to analyze multi-threaded applications. It is also important to note that all previously described tools except DrMemory are developed for Linux while we focused on Windows.

III. Problem Statement

Heap-based buffer overflows and UAF errors are the result of incorrect access to dynamically allocated memory. Heap-based bugs occur when program accessed out of the boundary of a dynamically allocated memory block. UAF is the case when the program performs erroneous access to the heap that has been already freed.

In Section II, we noted that heap overflows may be detected using special redzones that should be installed before and after each allocated memory block in the heap. We also need to remember the state (freed or in use) of these blocks to detect UAF. Thus, we need to handle each memory allocation/deallocation as well as perform boundary checking for each memory access in the SUT. Such an approach introduces significant overhead for the SUT. Experimental results have shown that the slowdown of DBI for each instruction may exceed x15-x20 on synthetic benchmarks [17], [11]. For widely-used complex products these values may be increased by several times which may significantly reduce possible application areas of such inefficient solution.

If we look at the ratio of the number of executed instructions in system DLLs to the number of executed instructions in the executables (Figure 1), it is possible to conclude that from 16% to 88% of instructions are executed in the system DLLs, while they are usually not a main purpose of testing.

On the other hand, errors may occur because of incorrect interaction between the SUT and the system DLLs. Therefore, we cannot fully exclude the system DLLs from instrumentation. The solution to this problem will be described further in Section V.

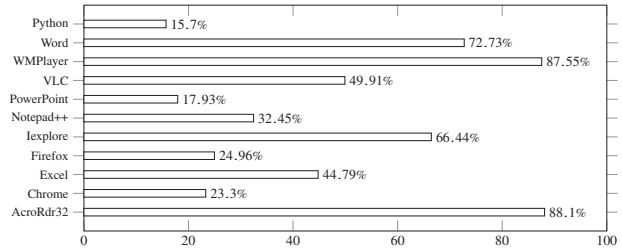


Fig. 1: A ratio of executed instructions in system DLLs (shown in the figure) to executed instructions in the main executables (along with shared libs). The data were collected using special tool implemented on top of the Intel Pin DBI framework. [23]

Moreover, it is sufficient to say that the mechanism of heap memory management in Windows is shared between several system libraries and implemented at different levels of WinAPI, sometimes without any official documentation. All of these introduce additional complexities for heap allocation/deallocation handling (see Section V for details).

IV. The general approach

Formal definition. We denote a trace of executed instructions as a system of ordered vectors $S = (s_0, s_1, ..., s_n)$, where n is a number of executed instructions, s_0 is an initial state, and s_n is a final state of a program.

We denote a state of the SUT after a single executed instruction as a vector:

$$s_i = \{P_i, M_i, Mfreed_i, RZbefore_i, RZafter_i\}$$

where P_i is the set of all pointers to the heap at s_i; M_i is the set of all allocated heap blocks at s_i; $Mfreed_i$ is the set of all freed heap blocks at s_i; $RZbefore_i$ is the set of all redzones (described further) installed before each memory block; $RZafter_i$ is the set of all redzones installed after each memory block.

We denote as γ a function that defines whether state s_i has a bug or not using the following expression:

$$\gamma(s_i) = \begin{cases} HoF, & if \quad P_i \cap RZafter_i \neq \emptyset \,\&\, write \\ HuF, & if \quad P_i \cap RZbefore_i \neq \emptyset \,\&\, write \\ HoR, & if \quad P_i \cap RZafter_i \neq \emptyset \,\&\, read \\ HuR, & if \quad P_i \cap RZbefore_i \neq \emptyset \,\&\, read \\ UAF, & if \quad P_i \cap Mfreed_i \neq \emptyset \end{cases} \tag{1}$$

Thus, successful bug detection involves memory access checking for each instrumented state in the SUT. The question is how to keep the actual state of $RZbefore_i$, $RZafter_i$, $Mfreed_i$ and M_i without high overhead for the instrumented application.

Since the memory in the heap is allocated with 8 bytes alignment, it is possible to map each 8 bytes of the heap to 1 byte stored in a special place in the SUT's memory. This special place called *shadow memory* and proposed early in [16]. To get an offset, it is enough to divide any memory

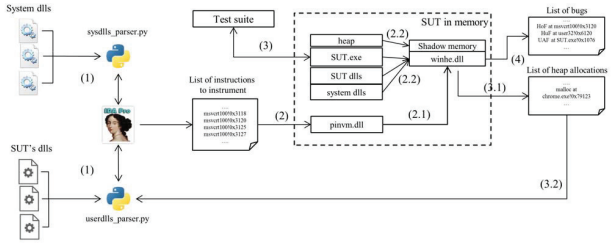


Fig. 2: The basic scheme of WinHeap Explorer

address by 8 (bit shift to the right by 3) and add the base address of the allocated shadow memory to the result.

V. Light-Weight Instrumentation

In this section, we will consider light-weight instrumentation approach as well as describe challenges that we have encountered during implementation of WinHeap Explorer.

WinHeap Explorer uses the Intel Pin DBI framework [23] to perform analysis of each instrumented state in the SUT. The framework is designed to allow external users to implement their custom instrumentation routines with different levels of granularity: (1) machine instruction level, (2) basic block level, (3) routine level and (4) image level. Pin starts the SUT suspended and then injects all required modules to begin instrumentation.

In WinHeap Explorer we perform instrumentation at levels (1), (3) and (4). At level (3), we handle WinAPI functions that perform heap memory allocation or deallocation. The special handlers are used to install/remove redzones and maintain information about the actual state of allocated/freed heap blocks. At level (4), we handle every new DLL that SUT loads in memory. Furthermore, we apply instrumentation at level (1), handling instructions that perform reads or writes in the heap. If an instruction attempts to access even one byte that correspond to some redzone or freed memory block, we signal an error and write it in a log file.

The basic scheme of WinHeap Explorer is represented in Figure 2. According to Figure 2, at step (1), the tool performs preliminary static analysis of the system and SUT DLLs using special plugins written in Python for IDA disassembler [32].

At (2), Pin loads the SUT in suspended state and injects pinvm.dll along with a list of instrumented instructions. At (2.1), pinvm.dll loads the instrumentation module winhe.dll. Winhe.dll allocates shadow memory and begins instrumentation (2.2). When a new DLL is loaded, the instrumentation module sequentially looks for each instruction in the DLL and decides whether the instruction should be instrumented or not based on the list of relative virtual addresses (RVA) obtained in the previous step. Then at (3), the test suite starts analysis. In parallel with heap access checking, the instrumentation module saves information about calls to heap allocation/deallocation procedures, which is passed back to the `userdlls_parser.py` plugin for additional instrumentation. This approach allows finding dynamic calls to heap allocation procedures and supports static analysis using information obtained during execution.

Algorithm 1: Custom DLLs instrumentation

```
1  function InstrumentCall (current_depth, func_ep);
     Input     : Current depth current_depth, , a function's
                 first address func_ep
     parameter: User defined max depth max_depth, user
                 defined list of functions (entry points) to
                 instrument funcs_list
     Output    : A list of instructions to instrument instr_list
2  if func_ep ∈ funcs_list then
3  |  return NULL;
4  else if current_depth ⩾ max_depth then
5  |  return NULL;
6  subcalls ← get list of subcalls;
7  foreach subcall do
8  |  ret_list = InstrumentCall(current_depth + 1,
   |    subcall_ep);
9  |  instr_list ∪ ret_list;
10 end
11 current_routine_instructions ← get list of instructions;
12 instr_list ∪ current_routine_instructions;
13 return instr_list;
```

System DLL instrumentation. At the first stage of analysis, WinHeap Explorer performs preliminary analysis of the system DLLs. The tool selects RVAs of the most potentially erroneous routines and saves them in the list along with information about instrumented DLLs.

In our research, we decided to select WinAPI functions (e.g. memcpy or fgets) that were identified by Microsoft Secure Development Lifecycle (SDL) [33] as unsecured and not recommended for use in modern software products. The full list of considered system DLLs is provided in the next section.

Shared DLL instrumentation. A similar approach cannot be applied to select routines from the DLLs where symbolic information is not available. To address this problem, we use Algorithm 1. We start analysis with routines that call functions from the Microsoft SDLC list [33]. We perform in depth analysis making a recursive search of subcalls for all routines found at the previous stage. While including additional routines may seem redundant, we have decided to include such functionality to cover situations when a bug appears in subroutines or callers as a result of incorrect memory handling in a mainly considered routine.

A. Technical Challenges

During development of WinHeap Explorer, we have encountered a number of technical challenges associated with specific features of heap management in Windows.

There are rare situations when two blocks of heap are allocated sequentially in the memory, thereby overlapping

TABLE I: Heap management functions in Windows

Dll name	API function	Low level API function in ntdll.dll
kernel32	HeapAlloc GlobalAlloc LocalAlloc	RtlAllocateHeap
msvcr*	malloc calloc operator new	
ole32	CoTaskMemAlloc	
kernel32	HeapReAlloc GlobalReAlloc LocalReAlloc	RtlReAllocateHeap
msvcr*	realloc	
ole32	CoTaskMemrealloc	
kernel32	HeapFree GlobalFree LocalFree	RtlFreeHeap
msvcr*	free operator delete	
ole32	CoTaskMemFree	

redzones. In such case, we do not install redzones to avoid false positives. Potentially, in such situations, the overlapped block may be reallocated; however, it may introduce additional artifacts and lead to unexpected behavior. This has been confirmed in our experiments when we tried to handle such situations; most of the widely-used complex applications, such as Firefox or Chrome browsers, crashed even during the initialization step.

The second challenge is a specific heap management mechanism in Windows that is spread across multiple system DLLs. Furthermore, MSDN [34] describes several classes of different WinAPI functions that can be used for heap management (Table 1).

As a result of reverse engineering, we figured out that each of these high level routines finally execute lower level routines from ntdll.dll called `RtlAllocateHeap`, `RtlReAllocateHeap` and `RtlFreeHeap`. This significantly facilitates instrumentation because we can focus on functions from ntdll.dll, without needing to handle high level wrappers like malloc or `CoTaskMemAlloc`. However, we still need to handle different flags passed to the `Rtl*` group of functions.

It should be noted that we inject instrumentation routines before and after each call to `RtlAllocateHeap`, `RtlReAllocateHeap`, `RtlFreeHeap` to obtain the size and base address of the allocated memory block. In the case of `RtlFreeHeap`, we insert instrumentation routines after a call to make sure that the memory block has been successfully freed and the function returns true.

VI. Experiments

In this section, we provide experimental evaluation of our approach. For all experiments we have deployed 2 virtual machines equipped with Windows XP SP3 x86 and Windows 7 SP1 x86, 4GB RAM and processor Intel Core i7-3630QM @2.40Ghz.

By default, WinHeap Explorer considers system DLLs listed in Table 2. Prior to the experiments, we conducted preliminary instrumentation of each of them and then shared the obtained database for all experiments.

TABLE II: List of supported system DLLs

System DLL name	Instrumented routines count	System DLL name	Instrumented routines count
user32	16	msvcrt	54
ntdll	35	kernel32	19
msvcr*	48	msvcp*	39

A. Evaluation

In our first experiment, we decided to verify that WinHeap Explorer can detect heap-based bugs in real-world applications. We randomly selected 10 working exploits from exploit-db [35] for the period from 2010 to 2016 year that use heap-based bugs in the available vulnerable applications for Windows 7 SP1 or Windows XP SP3. For each application, we manually conducted reverse engineering of the vulnerability to localize the address of the routine and certain machine instructions where incorrect memory access occurs (see description in Table 3). Furthermore, we instrumented all shared DLLs that are supplied with these applications. Links to exploits are available online at [1].

Each selected application has been executed within WinHeap Explorer launched with default parameters and then we used an exploit to trigger a bug. As a result, the system managed to detect all vulnerabilities with precision of concrete machine instruction and memory address in the heap.

`httpdx`. Due to incorrect input string length handling, the vulnerable http server writes 0x410 (using memcpy) bytes of memory to the buffer whose size is 0x400 bytes. WinHeap Explorer detects a heap out of bound write access at offset 0x39b60 in msvcr100.dll.

`Nplayer`. The error occurs due to incorrect external skin file processing. Nplayer calculates an offset in the memory based on the field value that is controlled by the user. As a result, it leads to heap overrun. WinHeap Explorer reports heap out of bound read access at 0x496f43.

`Microsoft Office Excel 2010`. The application uses a block of memory which has been previously freed by function AVrfpRtlFreeHeap in vfbasics.dll. WinHeap Explorer detects UAF at 0x30037cc5 in excel.exe.

`Core FTP Server`. The FTP server incorrectly handles external control commands. The application calls `sprintf(s, "200 MODE set to %s\n", command)` where the `command` size is 0xd4 and the size of `s` is 0x10. As a result, it causes the heap to overflow. WinHeap Explorer reports a heap out of bound write access at 0x7c90120e in ntdll.dll.

`gdi32.dll`. WinAPI function `ExtEscape` from gdi32 incorrectly handles its input size argument. We have implemented a special program to trigger the bug. Due to incorrect input parameter handling, function `ExtEscape` calls `memcpy(pDst, pSrc, 0xffffffff)` at offset equals 0x27a1a in gdi32.dll. Since the third argument is 0xffffffff, memcpy tries to rewrite 0xffffffff bytes of memory starting

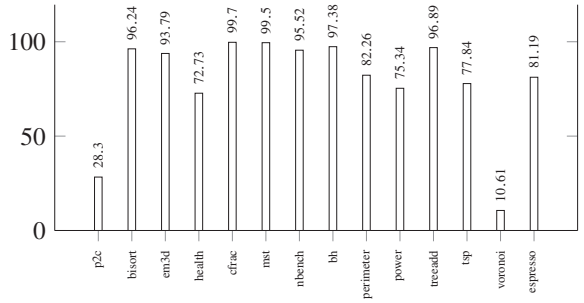


Fig. 3: A ratio of executed instructions in the main executable (shown in the figure) to system DLLs for selected benchmarks

at pDst which leads to heap corruption. WinHeap Explorer detects a heap out of bound write access at offset 0x34cf3 in msvcr100.dll.

`KingView 6.53 SCADA`. The vulnerability occurs in HistorySrv module of the KingView 6.53 SCADA system. The application allocates 0x100 bytes of memory and then erroneously copies 0x140 bytes due to incorrect parsing of the input string using `rep movsb`. WinHeap Explorer reports a heap out of bound write access at offset 0x1aef in HistorySrv.exe.

`MS Internet Explorer 11`. The application uses a heap block that has been previously freed in function CObjectElement (mshtml.dll). WinHeap Explorer detects a UAF bug at 0x6dfcc19b im mshtml.dll.

`RealPlayer 14.0.1.633`. Due to incorrect input data handling, it is possible to control the size of the allocated heap. The application incorrectly allocates 0x3ca bytes of memory while the input buffer size is controlled by the user and equals 0x500. WinHeap Explorer reports a heap out of bound write access in the function memcpy at 0x7855ae7a.

`Flare.exe 0.6`. The vulnerability occurs due to incorrect input argument handling. The application incorrectly allocates 0x100 bytes to handle the input argument, while the argument's length is controlled by the user and may be greater than 0x100 bytes. WinHeap Explorer reports a heap out of bound write access at 0x401dfe.

`MS Internet Explorer 11`. The application uses a previously freed (by MemoryProtection::HeapFree) block of memory in CSpliceTreeEngine::HandleRemovalMutations function. WinHeap Explorer reports a UAF at 0x67028aa8 in mshtml.dll.

The analysis shows that half of the bugs result from the use of unsafe and not-recommended WinAPI functions, such as memcpy and snprintf, as well as a low-level representation of the memcpy instruction `rep movsb`.

B. Runtime and Memory Overheads Evaluation

Since, to the best of our knowledge, DrMemory is the closest work to ours (see Section II), in this section, we use DrMemory to compare experimental results.

In our second experiment, we used a set of the following benchmarks: Nbench benchmarks suite [36], LockLess malloc

TABLE III: The details of vulnerable applications selected for experiment

Vulnerability type CVE/ISDB or ID in exploit-db Date	Name, app version OS Type	Vulnerable module Routine name Address
Remote HoF 20120 29.07.2012	httpdx 1.5.4 Win7 SP1	httpdx.exe memcpy 0x4088b8
Local HoR 11133 13.01.2010	Nplayer 1.2.0.7 Win7 SP1	Nplayer.exe no symbolic name 0x496f43
Local UAF CVE 2015-2523 16.09.2015	Microsoft Office Excel 2010 Win7 SP1	excel.exe no symbolic name 0x30037cc5
Remote HoF 39793 10.05.2016	CoreFTP Server 32bit build #587 Win7 SP1	coresrv.exe no symbolic name 0x4de56d
Local HoF CVE 2016-0170 17.05.2016	gdi32.dll Win7 SP1	gdi32.dll memcpy 0x77b54cf3
Remote HoF CVE 2011-0406 09.01.2011	KingView6.53 SCADA WinXP SP3	nettransdll.dll no symbolic name +0x1aef
Local UAF CVE 2015-6152 14.12.2015	Microsoft Internet Explorer 11 Win7 SP1	mshtml.dll CTreeNode:: ComputeFormatsHelper 0x6dfcc19b
Local HoF CVE 2011-1525 21.03.2011	RealPlayer 14.0.1.633 WinXP SP3	rvrender.dll memcpy 0x7855ae7a
Local HoF 40034 29.06.2016	Flare.exe 0.6 Win7 SP1	msvcrt.dll strcpy 0x75588d2
Local UAF 39699 15.04.2016	Microsoft Internet Explorer 11 Win7 SP1	mshtml.dll CSpliceTreeEngine:: HandleRemovalMutations 0x67028aa8

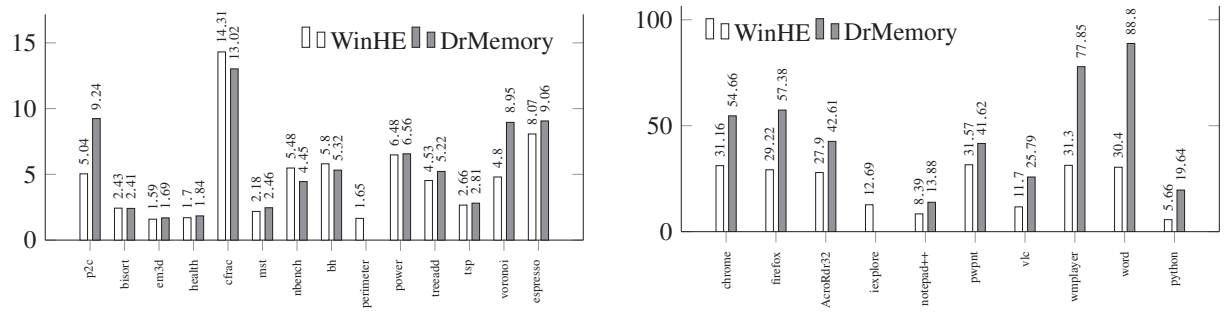
benchmarks [37][1], Olden benchmarks [38] along with 10 popular widely-used complex applications for Windows. We ran each test multiple times and present the median. Across all experiments, the 90th percentiles were typically within 10% and never more than 20% off the mean. To evaluate runtime and memory overheads, we used *Microsoft performance counters* [39]. For the 10 widely-used complex applications we conducted preliminary static instrumentation of all supplied DLLs. Since benchmarks do not have shared DLLs, we can only use instrumentation results of system DLLs for them.

There are several important parameters for WinHeap Explorer that may influence performance of the SUT.

Red zones size. The size of the redzones (by default 8 bytes) should not affect the performance of the SUT dramatically, however, large redzones may cause a collision between a redzone and an allocated heap when the heap is allocated in the same place as where redzones have been previously set. Such a situation requires special handling of a newly allocated block, which can slightly increase runtime overhead.

Instrumentation depth. The rate directly influences performance of the SUT. The more routines we include in the

[1] *gawk*, *ps*, *make* and *perl* benchmarks have been excluded because we did not manage to compile them for Windows.



(a) Benchmarks

(b) Real-world applications

Fig. 4: Runtime overhead

instrumentation the more runtime and memory overhead we have. The system spends time to instrument more instructions as well as to load a larger database in the memory. By default, the instrumentation depth is 2, however, it can be changed through the arguments provided by the tool.

1) Benchmarks: Before analysis of runtime overhead, we had a separate experiment to evaluate how intensive benchmarks use system DLLs (Figure 3). It turned out that most benchmarks use system DLL functionality very rarely. It means that the effect of light-weight instrumentation will be lower in these cases. However, we decided to provide experimental details to evaluate the general runtime overhead of WinHeap Explorer for such applications. We run WinHeap Explorer with default parameters for all experiments. Furthermore, to acquire the same functionality, we run DrMemory with the following parameters: *-no_check_handle_leaks*, *-no_check_leaks*, *-no_check_stack_access*, *-no_check_gdi*.

Figure 4(a)[2] shows that WinHeap Explorer runtime overhead for benchmarks ranges between x1.7-x14.31, which is comparable with results demonstrated by DrMemory. However, we encountered higher overhead for the *cfrac*, *nbench* and *bh* benchmarks in comparison with DrMemory. This can be explained by the higher overhead of the Intel Pin DBI framework itself in comparison with DynamoRIO. The detailed analysis of the frameworks performance may be found at [40]. It is not surprising to us that our system demonstrates better results on benchmarks (*voronoi*, *p2c*, *espresso*) that use system DLLs more intensively.

2) Real-world applications: Unfortunately, benchmarking of real-world applications is not trivial task because we need to correctly estimate the overhead in the cases when an application may run permanently in the memory (unlike specially designed benchmarks) and may have interaction with the user through GUI. To address this problem, we used a special technique for runtime and memory overhead estimations. It is possible to perform evaluation until a specific event such as a window popup, a network request, a file opening etc. It is important to say that an event is always accompanied with a specific libcall. We have manually reviewed and selected such events in a trace of libcalls obtained for each application (the details are available in our GitHub account [1]). To collect traces, we use DrLtrace tool supplied with the DynamoRIO framework [30]. Then, we manually patched libcalls so that it causes a program to finish execution immediately by calling the `ExitProcess` WinAPI function.

We consider runtime overheads for widely-used complex applications in Figure 4(b)[3]. Despite the fact that WinHeap Explorer demonstrates higher overhead than DrMemory for several benchmarks, in the real-world applications we have fixed lower overheads for all selected applications ranging between 24.2%-71.1%. WinHeap Explorer introduces lowest overheads for complex applications such as Chrome, Firefox, AcroRdr32, Wmplayer, Python and Word.

In parallel with runtime overhead, we evaluated the memory overhead of our approach for widely-used complex applications and benchmarks where WinHeap Explorer demonstrates the same level of memory overhead in comparison with DrMemory both for benchmarks and for widely-used complex applications (except Word and Python where we showed better results).

VII. Discussion

Currently, WinHeap Explorer handles only heap-based bugs, however, our approach may be expanded to support stack-based overflows as well as incorrect accesses to uninitialized memory. In general, we only need to map each stack and arbitrary memory address of the SUT in the shadow memory and add special markers for inaccessible memory zones.

While our approach of statically select potentially "interesting" routines may decrease code coverage of the SUT, we consider our solution as a reasonable trade-off between full instrumentation and testing without instrumentation at all.

Another limitation is related to the fact that redzones allow detection of heap-based bugs only when the application performs sequential out of bounds memory access, which is the main form of buffer overflow. WinHeap Explorer does not provide functionality to detect heap overflows in case of erroneous access to some other allocated heap block.

[2]Drmemory failed to execute the *perimeter* benchmark.

[3]DrMemory failed to instrument Internet Explorer launcher.

Despite the fact that shadow memory approach is free of false positives, there are bugs in the Intel Pin framework itself, as well as undocumented flags for RtlAllocateHeap, which can lead to false positives.

VIII. Conclusion

In this paper, we introduced an original approach and open-source proof-of-concept tool called WinHeap Explorer for heap-based bug detection in complex widely-used applications. We designed our system to be able to integrate into existing testing solutions with a low level of runtime and memory overheads. To achieve that, the tool uses light-weight binary instrumentation based on preliminary static analysis of the system and shared DLLs for potentially erroneous code paths, and dynamic instrumentation of selected machine instructions.

WinHeap Explorer has demonstrated lower runtime overheads ranging between 24.2%-71.1% and the same level of memory overhead comparing with existing solutions. Along with that, our experiments on vulnerable applications have proved WinHeap Explorer's ability to detect heap-based bugs in the machine code. In support of open science, we have published source code online at [1]. The tool is distributed under BSD license.

IX. Acknowledgments

The author would like to thank Cindy Eisner (IBM Research Israel) and Vyacheslav Zolotarev (Siberian State Aerospace University) for valuable comments that improved this research.

References

[1] "WinHeap Explorer source code repository," https://github.com/WinHeapExplorer/WinHeap-Explorer.
[2] "National Vulnerability Database. CVE and CCE Statistics Query Page," https://web.nvd.nist.gov/view/vuln/statistics.
[3] H. Etoh, "ProPolice: GCC Extension for Protecting Applications from Stack-smashing Attacks," *IBM (April 2003)*, 2003.
[4] Microsoft, "/GS (Buffer Security Check)," https://msdn.microsoft.com/en-us/library/8dbf701c.aspx.
[5] "US-CERT Alert (TA16-187A). Symantec and Norton Security Products Contain Critical Vulnerabilities," 2016, https://www.us-cert.gov/ncas/alerts/TA16-187A.
[6] "CVE 2016-1101. Adobe Flash – Heap Overflow in ATF Processing," 2016, https://www.us-cert.gov/ncas/alerts/TA16-187A.
[7] M. B. Amu, "Advanced Heap Spraying Techniques," 2010, https://www.owasp.org/index.php/File:OWASL_IL_2010_Jan_-_Moshe_Ben_Abu_-_Advanced_Heapspray.pdf.
[8] J. Feist, L. Mounier, and M.-L. Potet, "Statically Detecting Use After Free on Binary Code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.
[9] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward Large-Scale Vulnerability Discovery using Machine Learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 85–96.
[10] S. Rawat and L. Mounier, "Finding Buffer Overflow Inducing Loops in Binary Executables," in *2012 IEEE Sixth International Conference on Software Security and Reliability (SERE)*. IEEE, 2012, pp. 177–186.
[11] D. Bruening and Q. Zhao, "Practical Memory Checking with Dr. Memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011, pp. 213–223.
[12] M. Brünink, M. Süßkraut, and C. Fetzer, "Boundless Memory Allocations for Memory Safety and High Availability," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2011, pp. 13–24.
[13] F. C. Eigler, "Mudflap: Pointer Use Checking for C/C+," in *GCC Developers Summit*. Citeseer, 2003, p. 57.
[14] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight Bounds Checking," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 135–144.
[15] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," in *the Winter 1992 USENIX Conference*. Citeseer, 1991.
[16] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A Fast Address Sanity Checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.
[17] J. Seward and N. Nethercote, "Using Valgrind to Detect Undefined Value Errors with Bit-Precision," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.
[18] "Apple Corp. Guard Malloc Manual Page," 2009, https://developer.apple.com.
[19] Microsoft, "Gflags and PageGuards," https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561
[20] B. Perens, "Efence (3)," 1993.
[21] G. Watson, "Debug Malloc Library," *Letters Corp*, vol. 11, 1994.
[22] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.
[24] M. Shudrak and V. Zolotarev, "Improving fuzzing using software complexity metrics," in *International Conference on Information Security and Cryptology*. Springer, 2015, pp. 246–261.
[25] D. Duran, D. Weston, and M. Miller, "Targeted taint driven fuzzing using software metrics," 2011.
[26] R. L. Seagle Jr, "A framework for file format fuzzing with genetic algorithms," Ph.D. dissertation, 2012.
[27] C. Cifuentes and K. J. Gough, "Decompilation of Binary Programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.
[28] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 260–275.
[29] G. Balakrishnan and T. Reps, "WYSINWYX: What You See is not What You eXecute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, p. 23, 2010.
[30] D. L. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.
[31] "The Chromium Project," https://www.chromium.org.
[32] "IDA Dissasembler," https://www.hex-rays.com/index.shtml.
[33] M. Howard, "Microsoft Secure Development Lifecycle. List of Banned Syscalls," 2011, https://msdn.microsoft.com/en-us/library/bb288454.aspx.
[34] "Managing Heap Memory," https://msdn.microsoft.com/en-us/library/ms810603.aspx.
[35] "Offensive Security Exploit Database Archive," https://www.exploit-db.com.
[36] "Nbench benchmarks suite," https://github.com/Microsoft/test-suite/tree/master/MultiSource/Benchmarks/nbench.
[37] "Lockless malloc benchmarks," http://locklessinc.com.
[38] "Olden benchmarks," http://www.martincarlisle.com/olden.html.
[39] "Microsoft performance counters," https://msdn.microsoft.com/en-us/library/windows/desktop/aa373083
[40] "Building Dynamic Tool with DynamoRIO on x86 and ARM," 2016, https://github.com/DynamoRIO/dynamorio/releases/download/release_6_1_0/DynamoRIO-tutorial-mar2016.pdf.