Positive Hack Days 2017

Exploring billion states of a program like a pro. How to cook your own fast and scalable DBI-based security tool.

A case study.

Maksim Shudrak

PhD, Security Researcher

IBM Research Israel

Haifa Labs

Part 0. Motivation or why do we need this talk?

In 2012, Dmitry Evdokimov conducted a talk at Positive Hack Days about code instrumentation. He provided a various specter of existent techniques for code instrumentation including source code, byte code and machine code and he briefly described dynamic binary instrumentation (DBI) as well. In my talk, I would like to focus on DBI exclusively, delve deeper in this topic, demonstrate a power of this technique based on real practical examples as well as consider typical problems of its applications for "industrial" tasks.

Speaking in general, in my opinion this topic is insufficiently covered by the security community. There are very good presentations given at well-known international conferences (for example [1-4]) which describe different DBI-based security tools and solutions. However, they do not give idea about all capabilities of this technique and the most important challenges that an author may face working on his/her own tool. In my presentation, I would like to reduce this gap.

Part 1. Introduction

Nowadays, we have several robust and multifunctional DBI-frameworks for the most popular OS (Windows, Linux, Mac OS, Linux) supported by industry and science communities. DBI is a technique of analyzing the behavior of a binary application at runtime through the injection of instrumentation code.

The basic idea of DBI may be represented at Figure 1. Per Figure 1, the first stages look like classical DLL-injection. An application starts in suspended state, the core DBI-framework's DLL is injected in a target and then control flow is redirected into this DLL. At the next step, the library code performs environment initialization, loads all required additional modules and then loads user-written DLL to begin instrumentation.

After that, at stages 4-6, it begins sequential translation of each basic block (a sequence of instructions without conditional or unconditional jumps) into intermediate representation. Depending on user's requirements (defined in a user DLL), framework performs injection of instrumentation code (marked gray in our example) and then second translation back into machine code for execution in so-called "code cache". Then it performs address calculation of a next executed basic block - > translation -> instrumentation -> translation back into machine code -> execution -> calculation of a next basic block and so on until the program finish.

Even though the general idea seems relatively trivial, correct and effective (in terms of reliability, transparency and performance) implementation of this approach is a complex engineering task that considered in details in these works [5-7].

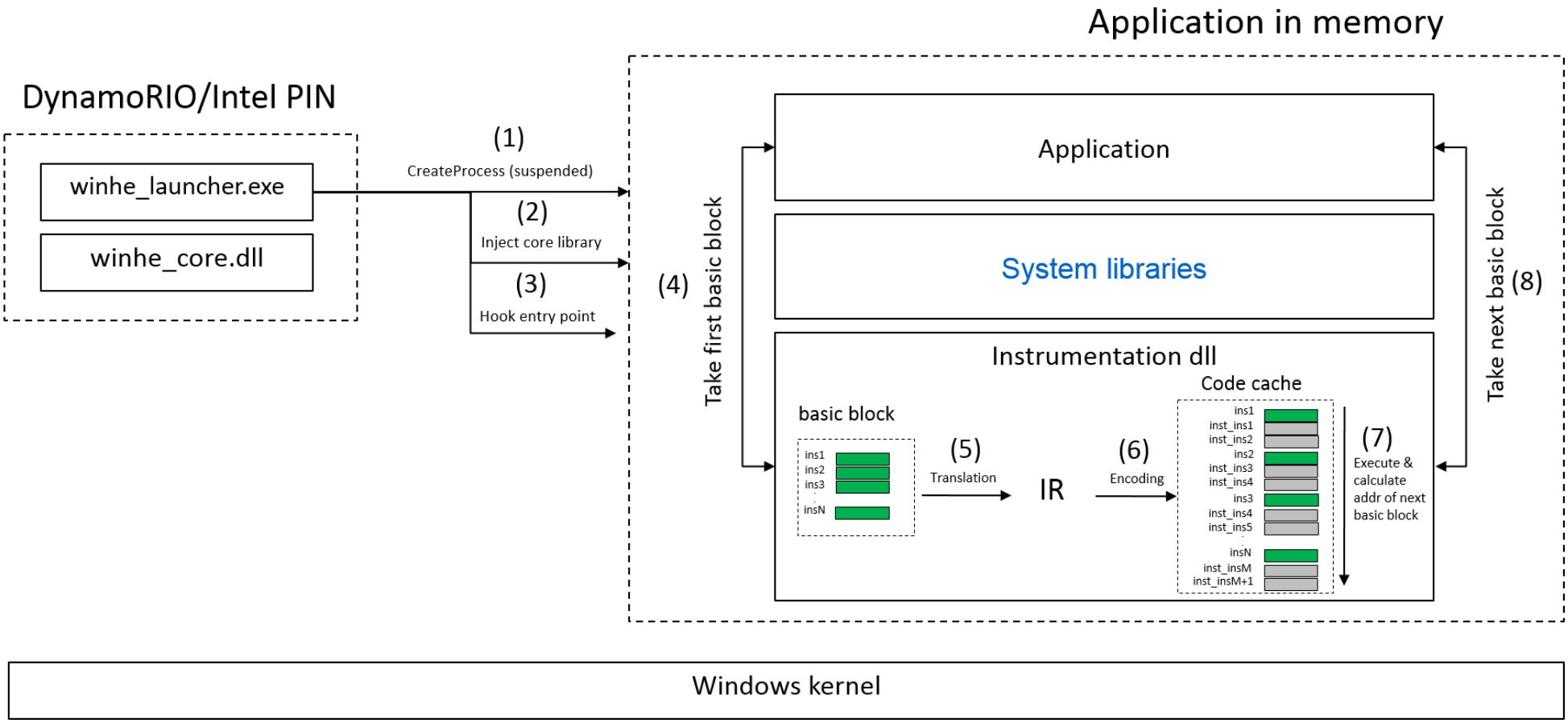

Figure 1. DBI -framework architecture and step-by-step description

Nowadays DBI is successfully applied in the following fields:

- Bugs and vulnerabilities hunting, especially in a case of source code absence [8]. DBI-based solutions are successfully integrated in secure development lifecycle (SDL) [9], used to support fuzzing [10] as well as for automatic software bug exploitation detection [11-12].
- Malware analysis [13]. Despite the possibility that instrumented application might easily detect a fact of instrumentation as shown in the following researches [15-16], DBI allows to transparently trace an application library and system calls [14] and may be used for automatic malware unpacking [3-4].
- Reverse-engineering. DBI successfully uses for taint-tracking [17], control-flow graph visualization [18] as well as debugging [19].
- In many others fields that are not directly connected with security but very important for software development (performance evaluation, memory leak detection, optimization etc.).

Part 2. WinHeap Explorer. An example of DBI application.

Modern DBI-frameworks have well-developed API and allows to perform instrumentation at different layers of granularity: for each instruction, basic block, function, trace, module or specific event related to instrumented program (e.g. program exit, child process creation etc.). Based on this rich API, author has implemented a tool for transparent and efficient heap-based bug detection in machine code called WinHeap Explorer.

WinHeap Explorer is a tool for Windows that allows to detect heap out of boundary accesses and use-after-free bugs during software execution. WinHeap Explorer is based on Intel PIN. The most significant advantage of this tool is that unlike existent systems (DrMemory/Valgrind) it allows to perform light-weight instrumentation and thereby reduce runtime overheads introduced by instrumentation (in average by 24% - 71% depending on application as shown in Figure 2).

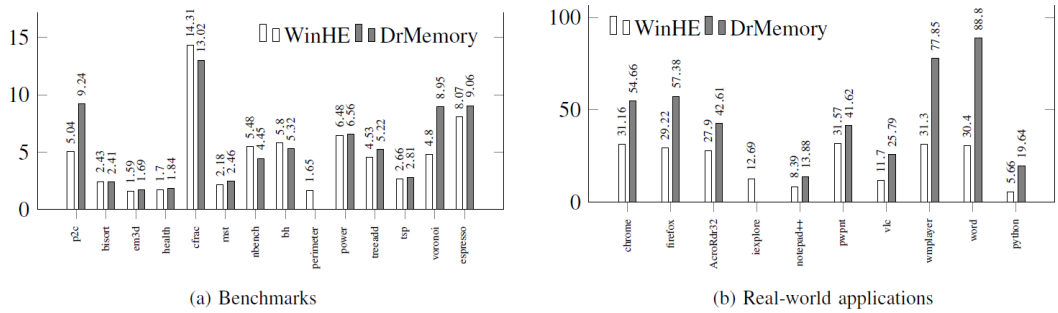

Figure 2. Runtime overheads comparison. Y-axis is an overhead.

The basic idea of this tool is to perform boundaries checking for all accesses to memory that application perform during execution. To achieve this, we set up special flags (called *redzones*) before and after each allocated memory block. If an application reads or writes memory where redzones are installed such situation is considered as a bug. Sounds easy while in practice there are several serious difficulties.

In WinHeap Explorer, we perform instrumentation for each instruction and in case of a new DLL loading. We also perform instrumentation for each library call that allocates/frees memory in a heap. The main goal of instrumentation is to implement the approach described above. (setup/remove redzones for each allocated memory block in a heap).

Then we perform instrumentation of each executing instruction that accesses heap. If at least one byte which an instruction accesses is inside redzone, WinHeap Exlorer logs information about a bug in the following format:

```
[HEAP OVERFLOW] at 0x496f43, writing out of bound to 0x37110f; heap has been allocated at 0x495f13, size = 0x300, flags = 0}
```

During implementation of this idea, I have encountered the following two problems:

1) Windows memory management mechanism is a complex undocumented process that spreads across various system libraries and implemented at different WinAPI levels, sometimes without official documentation including dozens of undocumented flags that might significantly change heap management mechanism.
2) The number of allocated/freed memory blocks in modern programs may exceed millions and the number of accesses to them may reach billions attempts. To handle this situation, it is necessary to somehow store information about installed redzones and effectively provide access to it in case when an instruction reads or writes in a heap.

To resolve the first problem, I've performed a detail reverse-engineering of Windows system DLLs related to heap management and concluded that we do not need to perform instrumentation of all WinAPI calls (related to heap allocation or deallocation), it is enough to apply instrumentation only at `ntdll.dll` (Table 1) because in any case high-level WinAPI calls lead to `RtlAllocateHeap, RtlReAllocateHeap, RtlFreeHeap`.

Table 1. Heap-management in Windows.

Feature	Dll name	API – function	Corresponding low-level API function from ntdll.dll
Memory allocation	kernel32.dll	HeapAlloc GlobalAlloc LocalAlloc	RtlAllocateHeap
	msvcr*.dll	malloc calloc operator new []	
	ole32.dll	CoTaskMemAlloc	
Memory reallocation	kernel32.dll	HeapReAlloc GlobalReAlloc LocalReAlloc	RtlReAllocateHeap
	msvcr*.dll	realloc	
	ole32.dll	CoTaskMemrealloc	
Freeing memory	kernel32.dll	HeapFree GlobalFree LocalFree	RtlFreeHeap
	msvcr*.dll	free operator delete []	
	ole32.dll	CoTaskMemFree	

To address the second problem, in the first iteration, I have implemented a straight-forward solution. Information about redzones and allocated heap blocks was stored in a hash table. Unfortunately, in terms of DBI, traditional in Computer Science data structures that provides high access speed are still not good enough, introduce tremendous overheads and cannot be used to instrument industrial applications (experiments have shown that for example launch of Microsoft Word takes in average 17 minutes with x500 memory overheads if we use such approach).

To address this problem, we have decided to use shadow memory approach. Shadow memory is a technique used to track and store information on computer memory used by a program during its execution. Let us describe this approach in Figure 3.

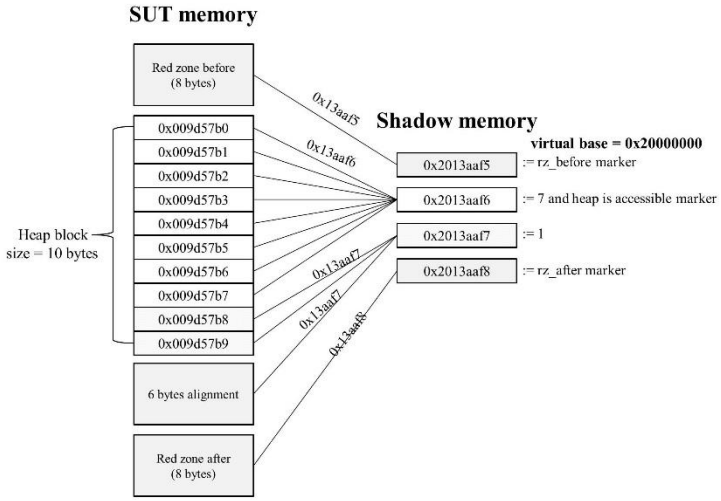

Figure 3. The allocated memory block mapped to the shadow memory. Each 8 bytes can be mapped to one byte of the shadow memory. SUT – software under test.

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 of the shadow memory as shown in Figure 3. To get an offset in the shadow memory, it is enough to divide any memory address by 8 (bit shift to the right by 3) and add the base address of the allocated shadow memory to the result.

To handle a new block of memory, it is necessary to perform the following steps:

1) get the size of the allocated memory block;
2) sequentially divide each allocated address by 8 and store the remainder plus a `heap_is_accessible` marker in the corresponding shadow bytes;
3) map 8 bytes before the allocated memory block to shadow memory and store a special marker (`rz_before`) in the corresponding shadow byte;
4) get the size of an alignment using the following expression: `size_of_align` = 8 - (`size_of_block`%8);
5) map 8 bytes after the allocated memory block plus the size of alignment to shadow memory and store a special marker (`rz_after`) in the corresponding shadow byte. It should be noted that the special markers should be greater than 7 to avoid collision with a remainder.

The reverse operation is performed when SUT frees the allocated heap block. WinHeap Explorer removes redzones before and after and marks all corresponding shadow memory as freed. If an instrumented machine instruction accesses a memory address which corresponds to shadow memory marked as `rz_after, rz_before, heap_freed` or the remainder stored in the shadow memory is less than the remainder of a memory address (e.g. `0x009d57ba`), such a situation is considered as a bug.

This approach allows to significantly reduce runtime overheads (by 40-65 times) introduced by our instrumentation.

During implementation of WinHeap Explorer, I have decided to go further and add an ability to perform partial instrumentation only for those parts of the code that are most interesting for researcher, thereby reducing runtime overheads even more. It may be useful when the tool is integrated with fuzzer or performance of software testing plays a significant role. For example, when a researcher doesn't want to instrument system dlls (Figure 4) and wants to focus on bugs hunting in a target application.

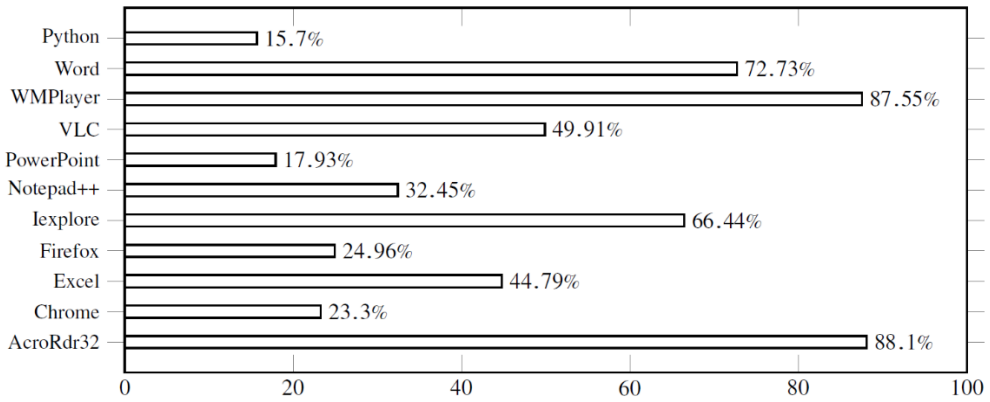

Figure 4. A ratio of executed instructions in system DLLs (shown in the figure) to executed instructions in the main executables (along with shared libs). The data were collected using a special tool implemented on top of the Intel Pin DBI framework.

On the other hand, errors may occur because of incorrect interaction between the target application and the system DLLs (for example in case of erroneous `memcpy` call, a bug occurs in a system DLL as shown in Listing 1). Therefore, we cannot fully exclude the system DLLs from instrumentation.

```
1   int some_function() {
2   <function prolog is skipped>
3   FILE *f = fopen("some_file.txt", "r");
4   004112E9  push      offset string "r"
5   004112EE  push      offset string "some_file.txt"
6   004112F3  call      dword ptr fopen
7   004112F9  add       esp,8
8   004112FC  mov       dword ptr [f],eax
9   char *buf_a = (char *)malloc(0x300);
10  004112FF  push      300h
11  00411304  call      dword ptr malloc
12  0041130A  add       esp,4
13  0041130D  mov       dword ptr [buf_a],eax
14  char *buf_b = (char *)malloc(0x30);
15  00411310  push      30h
16  00411312  call      dword ptr malloc
17  00411318  add       esp,4
18  0041131B  mov       dword ptr [buf_b],eax
19  fgets(buf_a, 0x100, f);
20  0041131E  mov       eax,dword ptr [f]
21  00411321  push      eax
22  00411322  push      100h
23  00411327  mov       ecx,dword ptr [buf_a]
24  0041132A  push      ecx
25  0041132B  call      dword ptr fgets
26  00411331  add       esp,0Ch
27  memcpy(buf_b, buf_a, 0x100);
28  00411334  push      100h
29  00411339  mov       eax,dword ptr [buf_a]
30  0041133C  push      eax
31  0041133D  mov       ecx,dword ptr [buf_b]
32  00411340  push      ecx
33  00411341  call      memcpy
34  631DC970  <function prolog is skipped>
35  631DC9C7  rep movs dword ptr [edi],dword ptr [esi]
36  ;edi -> buf_b, esi -> buf_a
37  631DC9XX  <function epilog is skipped>
38  00411346  add       esp,0Ch
39  00411349  xor       eax,eax
40  }
```

Listing 1. An example of the heap overflow in interaction with the system DLL. The overflow would occur at `0x631dc0c9` in the msvcr100.dll when instruction `rep movs` writes out of bound of `buf_b`.

Using this idea, the special script based on IDAPython has been implemented (available in WinHeap Explorer repository). The script allows to include only "interesting" library calls in terms of security testing. In this research, I decided to select WinAPI functions that were identified by Microsoft SDL [20] as unsecured and not recommended for use in modern software products (e.g. `memcpy` or `strcat`).

A similar approach cannot be applied to select routines from the DLLs where symbolic information is not available. To address this problem, we use a special algorithm shown in Listing 2. Thus, we start analysis with routines that call functions from the Microsoft SDL list. At the next step, we perform in depth analysis making a recursive search of subcalls for all routines found at the previous stage. At Line 22, we additionally collect data about callers. 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.

```
def instrument_subcalls(subcall, current_depth, list_of_routines):
    global max_depth
    list_to_return = list()
    if routine in list_of_routines:
        return null
    if current_depth >= max_depth:
        return null
    list_of_subcalls = get_list_of_subcalls(subcall)
    for subcall in list_of_subcalls:
        list_to_return.append(instrument_subcalls(subcall, current_depth + 1, list_of_routines)

    return list_to_return


max_depth := <user specified value>
list_of_routines = get_list_of_interesting_routines()
final_list = list()
for routine in list_of_routines
    list_of_subcalls = get_list_of_subcalls(routine)
for subcall in list_of_subcalls:
    final_list.append(instrument_subcalls(subcall, 0, list_of_routines))
final_list.append(get_list_of_callers(depth))
```

Listing 2. User dlls instrumentation. Implementation in Python.

Thus, the main scheme of WinExplorer integration in the process of software security testing is shown in Figure 5.

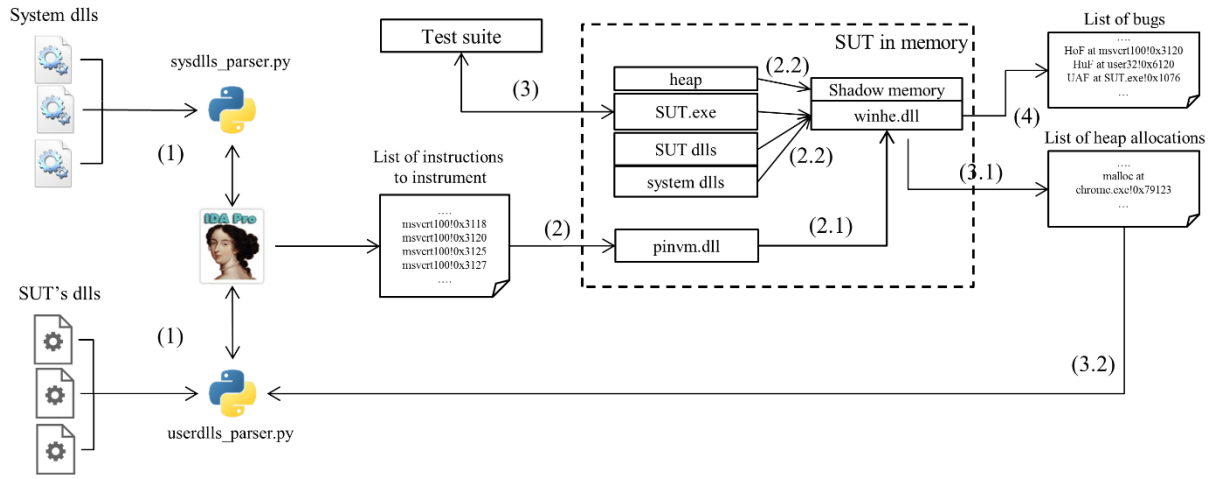

Figure 5. An example of WinHeap Explorer integration within software testing. IDAPython scripts are used to decrease amount of code to instrument SUT – software under test. Test suite – any framework for software testing including fuzzers.

Part 3. Examples

It was mentioned above that special IDAPython script is used to search for functions in system DLLs that are necessary to instrument. To collect these functions, the following commands might be used:

```
sysdlls_parser.py [path to system dll]
user_dlls_parser.py [path to user dll]
```

The scripts will automatically generate a list of instructions from selected routines that are necessary to instrument (IDA with IDAPython is required).

To start using WinHeap Explorer, it is enough to execute the following command:

```
pin.exe –t winhe.dll –d [path] [app_name]
Arguments:
-d [path] – a path to a file with instruction addresses for partial instrumentation,
the file should be produced by IDAPython scripts.
-redzones_size [bytes #] – redzones size.
-o <output path> - log file.
```

Let's consider several examples (*during the presentation I will expand this list, show demo, and provide a detail step-by-step analysis of the process of vulnerability detection in WinHeap Explorer to demonstrate how DBI works internally*) of heap overflows and use-after-free bugs detection using WinHeap Explorer:

- Httpdx 1.5.4. Remote Heap Overflow. 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.
- Microsoft Office Excel 2010 (CVE 2015-2523) BIFFRecord Use-After-Free. The application uses a block of memory which has been previously freed by function `AVrfpRtlFreeHeap` in vfbasics.dll. WinHeap Explorer detects use-after-free at `0x30037cc5` in excel.exe.
- gdi32.dll (CVE 2016-0170) Heap-based Buffer Overflow in `ExtExcape`. WinAPI function `ExtEscape` from gdi32.dll incorrectly handles input argument responsible for the size of memory to be copied. As a result, the function calls `memcpy(pDst, pSrc, size)` where argument size is controlled by user. WinHeap Explorer detects a heap out of bound write access at offset `0x34cf3` in msvcr100.dll.
- KingView 6.53 SCADA System (CVE 2011-0406) HMI Remote Heap-Overflow. The vulnerability occurs in HistorySrv module. 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.
- RealPlayer 14.0.1.633 (CVE-2011-1525) Local Heap Overflow. 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`.

Conclusion

DBI is a powerful technique that allows to perform a detailed analysis of machine code. There are a number of solutions based on DBI that successfully uses for reverse-engineering, malware analysis, vulnerabilities detection as well as in many other fields of Computer Science. In the white-paper, we propose an Intel PIN-based tool WinHeap Explorer for transparent and efficient heap-based bug detection that allows to perform light-weight instrumentation. I have shown the difficulties that I faced during implementation of WinHeap Explorer and demonstrated several examples of use-after-free and heap-based bugs detected using WinHeap Explorer.

References

[1] D. Dorsey. Why Don't You Just Tell Me Where the ROP Isn't Suppose to Go. DefCon 22.
[2] S. Choi. API Deobfuscator. Resolving Obfuscated API Functions in Modern Packers. Black Hat USA 2015.
[3] S. Mariani, L. Fontana, F. Gritti, S. D'Alessio. PinDemonium. a DBI-based generic unpacker for Windows executables. BlackHat USA 2016.
[4] J. Wook Oh. Vulnerability Analysis and Practical Data Flow Analysis & Visualization. CanSecWest 2012.
[5] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*. Vol. 42. No. 6. ACM, 2007.
[6] Luk, Chi-Keung, et al. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*. Vol. 40. No. 6. ACM, 2005.
[7] D. Bruening. Efficient, transparent, and comprehensive runtime code manipulation. Diss. Massachusetts Institute of Technology, 2004.
[8] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference* (pp. 309-318).
[9] P. Godefroid, M. Levin, D. Molnar. Automated Whitebox Fuzz Testing. *NDSS*. Vol. 8. 2008.
[10] M. Moroz. Scalable and Effective Fuzzing of Google Chrome. Positive Hack Days 2016.
[11] L. Davi, S. Ahmad-Reza, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. *Proceedings of the 2009 ACM workshop on Scalable trusted computing*. ACM, 2009.
[12] Z. Qin, A. Feng, et al. Lift: A low-overhead practical information flow tracking system for detecting security attacks. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006.
[14] Y. Heng, et al. Panorama: capturing system-wide information flow for malware detection and analysis. *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.
[13] https://github.com/DynamoRIO/drmemory/tree/master/drltrace
[15] K. Sun, X. Li, Y. Ou. Break Out of the Truman Show. Active Detection and Escape of Dynamic Binary Instrumentation. Blackhat Asia 2016
[16] F. Falcon and N. Riva. Detecting Binary Instrumentation Frameworks. ReCon – 2012.
[17] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2008.
[18] https://software.intel.com/en-us/articles/pintool-dcfg
[19] Q. Zhao, et al. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. *International Conference on Compiler Construction*. Springer Berlin Heidelberg, 2008.
[20] https://msdn.microsoft.com/en-us/library/bb288454.aspx
[21]https://github.com/WinHeapExplorer/WinHeap-Explorer/blob/master/Links%20to%20exploits