REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* 11/02/2020	**2. REPORT TYPE** final	**3. DATES COVERED** *(From - To)* Jan 2018 - Dec 2019

4. TITLE AND SUBTITLE Model-based Fuzzing for Finding Kernel Vulnerabilities	**5a. CONTRACT NUMBER** N00014-18-1-2024
	5b. GRANT NUMBER GRANT12472381
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Cha, Sang Kil	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, South Korea	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 875 N. Randolph Street Suite 1425 Arlington, VA 22203-1995	**10. SPONSOR/MONITOR'S ACRONYM(S)** ONR
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution is Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Kernel vulnerabilities are significant threats to computer security as attackers use them to obtain unauthorized root privilege and bypass security mitigations. In this project, we extend IMF, a state-of-the-art kernel fuzzing technique on macOS, to find vulnerabilities on Windows kernel. Unlike other OSes, Windows has numerous undocumented system calls, which make it difficult to generate a valid sequence of system calls for fuzzing. We propose a novel way to analyze Windows kernel binaries to figure out API specifications of undocumented system calls. We then use the inferred API specifications to generate a C program that automatically fuzzes Windows kernel by calling a randomly generated sequence of system calls.

15. SUBJECT TERMS
kernel; fuzzing; Windows fuzzing; kernel vulnerabilities

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Sang Kil Cha
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UU		**19b. TELEPHONE NUMBER** *(Include area code)* +82-42-350-3569

Model-based Fuzzing for Finding Kernel Vulnerabilities

(Final Report)

Sang Kil Cha
sangkilc@kaist.ac.kr

February 23, 2020

Abstract

Despite its security impact, Windows kernel fuzzing has gained less attention in research community. Windows kernel fuzzing is challenging as its kernel source code is not available, and many of the system calls are undocumented. In this research project, we build our own binary analyzer for Windows kernel, which infers dependence relationship between system call parameters as well as their data types. The inferred information is then used to create a C program that randomly fuzzes the Windows kernel code by invoking various combinations of system calls. Our system found several real-world kernel panic bugs, and we reported them to Microsoft.

1 Introduction

Kernel vulnerabilities are significant threats to computer security as attackers use them to obtain unauthorized root privilege and bypass security mitigation [24]. Accordingly, security exploits that involve local privilege escalation get higher prices [26], and the number of kernel CVEs is sharply increasing every year.

Therefore, many researchers including ourselves have been studying ways to automatically find kernel vulnerabilities [7, 21, 22]. Our previous work [7] on finding kernel vulnerabilities, which we referred to as IMF, had presented a novel technique that leverages an inferred dependence model between system API functions. The key intuition is to produce valid-looking sequences of system calls based on the model to explore deep kernel code without raising apparent type errors. IMF found more than 50 previously undisclosed vulnerabilities, all of which were reported to Apple.

Although IMF was successful in finding vulnerabilities on macOS, it is not straightforward to apply the same technique on Windows for several reasons. First, Microsoft does *not* provide enough documentation for their system calls. There are several known function prototypes for a few system calls, but the kernel has numerous undocumented system calls.

This means it is not even clear which system calls we should monitor. Second, system calls in Windows are dependent to each other, and it is not straightforward to group them into an independent set. In macOS, for instance, one could focus on the IOKit library functions as they are independently usable without requiring other external libraries. Finally, parameter types for Windows system calls are not unique. For example, the second parameter of LoadCursor function is known to be `const char*`, but it can become an integer depending on the value of the first parameter. All these challenges make it radically difficult to apply the IMF technique on Windows.

In this research project, we present a novel way to fuzz Windows kernel, and discuss the design and implementation of our system. Our system has found various realistic Windows kernel panics, and we are going to submit our paper to CCS this year, which is one of the top security conferences.

2 Background

2.1 Windows System Calls

Since IMF fuzzes system calls, it is imperative to understand how Windows system calls work. Windows uses two libraries `ntdll.dll` and `win32u.dll` to invoke system calls: both libraries provide system call stubs such as `NtCreateFile` and `NtSetEvent`. Although possible, programmers normally do not call such functions directly. Instead they write a program with regular Windows API functions, which internally call such system call stubs. This is mainly because Microsoft provides well-written documentation for Windows API functions [15], but not for Windows system calls. Specifically, it is known that there are various distinct system calls that can be invoked either from `ntdll.dll` or `win32u.dll`, but only a hundred of them have public description in MSDN [16]. Reverse engineers and security researchers have published their findings in an ad-hoc manner [2, 9, 10], but it is not enough to fully understand the system calls.

2.2 Existing Windows Fuzzers

Unlike user-land fuzzing, kernel fuzzing mostly focuses on invoking a series of system calls as system calls form a trust boundary between user code and kernel code. For example, if we can crash the target kernel by invoking system calls with specific parameters, that means user code can perform a denial-of-service on kernel code. Of course, one can further craft the way of invoking system calls to obtain unauthorized root access.

We can categorize existing Windows kernel fuzzers into three main groups based on their target. First, there are general-purpose fuzzers that target any Windows system calls [5, 11, 14, 17, 19]. Second, there are fuzzers focusing on fuzzing IOCTL (Input and Output Control) interface [4, 13, 18, 20]. Finally, there are fuzzers that aims to test a specific file format such as font fuzzer [12]. All these fuzzers are more or less the same in that they fuzz system API functions even though their target may differ. One notable exception is CAB-Fuzz [13], which leverages dynamic symbolic execution to explore kernel code. However, it suffers from the traditional path explosion problem [3] as it explores one execution path at a time.

3 Motivation

Windows kernel is an attractive target for attackers as it is the most popular OS for desktop and laptop computers. Indeed, most malicious applications are targeting Windows OS today [1]. Therefore, finding security vulnerabilities from Windows kernel and debugging them is crucial for software security.

Most existing Windows kernel fuzzers are *not* systematic in that they try to fuzz system API functions with random parameter values. However, simply passing random parameter values to system API functions is not enough to explore deep kernel code as they are likely to return an error. We should aware of the relationship between API functions to execute deep kernel code. To see why understanding such a relationship matters, consider an example program that triggers CVE-2017-8487 shown in Figure 1.

In this example code, an attacker passes an IOCTL request to `\Device\KsecDD` in order to leak uninitialized kernel memory values. With this vulnerability, the attacker can bypass kernel address space layout randomization. There are five API functions used in the example, and they are dependent on each other. For example, `RtlInitUnicodeString` should always proceed `NtOpenFile` in order to specify the device name, and `InitializeObjectAttributes` function initializes `objattr` used as the third parameter of `NtOpenFile`.

Suppose we are given the five functions, and we want to design a fuzzer that randomly call the functions with random parameter values. It is not straightforward to find the vulnerability with our fuzzer because (1) it needs to correctly order the function calls, and (2) it needs to correctly connect output values of a function to input values of other functions. We call such relationships as *ordering dependence* and *value dependence*, respectively, in IMF [7].

Unfortunately, unlike macOS, we are not aware of the specification of most of the Windows system calls. In other words, there is no documentation on how we call such system calls, as they are internally used by the system, but not by the users. Therefore, we cannot directly leverage the design of IMF [7], which assumes the knowledge of API specification. Instead, we should analyze the Windows kernel binary in order to figure out such information from scratch. The very first step, therefore, is to study the organization of the current Windows system calls.

4 Study on Windows System Calls

Unlike macOS, Windows system calls are largely unknown, and thus, we do *not* even know which system calls to instrument. Therefore, we first studied how many system calls are available on Windows. According to [25], system call stubs are defined in either `ntdll.dll` or `win32u.dll`.

With manual inspection, we found that system call stubs have a certain syntactic pattern for each architecture. For instance, Figure 2 presents a stub for `NtAccessCheck` system call for both x86 and x86-64. Notice we first set the system call number to `eax` with the `mov` instruction in both cases, and call a function that raises a software interrupt with the `sysenter` or `int` instruction. On x86-64 Windows, it uses WOW64 (Windows on Windows 64) to call 32-bit system calls.

By looking at the stub code, we can figure out several useful information about system

```c
HANDLE hksecdd;
OBJECT_ATTRIBUTES objattr;
UNICODE_STRING ksecdd_name;
IO_STATUS_BLOCK iob;
NTSTATUS st;

RtlInitUnicodeString(&ksecdd_name, L"\\Device\\KsecDD");
InitializeObjectAttributes(&objattr, &ksecdd_name, 0, NULL, 0);
st = NtOpenFile(&hksecdd,
        FILE_READ_DATA | FILE_WRITE_DATA | SYNCHRONIZE,
        &objattr,
        &iob,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        FILE_SYNCHRONOUS_IO_NONALERT);

if (!NT_SUCCESS(st)) {
    printf("NtOpenFile failed, %x\n", st);
    return 1;
}

BYTE InputBuffer[] = "\x4d\x3c\x2b\x1a\x00\x00\x02\x00\xff\xff\xff\xff\x00\x00\x00\x00\x20\x00\x00\x00\xff"
                     "\xff\xff\xff\x01\x00\x00\x00\x02\x00\x00\x00\x33\x00\x44\x00\x45\x00\x53\x00\x00\x00";
BYTE OutputBuffer[0x200] = { /* zero padding */ };
DWORD BytesReturned = 0;
if (!DeviceIoControl(hksecdd, 0x390400, InputBuffer, sizeof(InputBuffer),
                     OutputBuffer, sizeof(OutputBuffer), &BytesReturned, NULL))
{
    printf("DeviceIoControl failed, %d\n", GetLastError());
    CloseHandle(hksecdd);
    return 1;
}
```

Figure 1: An example triggering CVE-2017-8487 on Windows.

Idx	**DLL file**	**Windows Version**	**Number of Matches**
1	ntdll.dll	10.0.17134.1	456
2	win32u.dll	10.0.17134.1	1,132

Table 1: Windows 10 syscall stubs we found.

calls. First, we know which syscall number is used in the stub. Second, by looking at the operand value of the `retn` instruction, we can deduce the number of parameters used for each system call. Finally, we can see the name of the system call by looking at the debugging symbol of the stub.

We analyzed all the DLL files installed in a default Windows 10 environment to find the syscall stub pattern. Specifically, we enumerated functions in each DLL file and check if each function matches the patterns shown in Figure 2 while allowing the green parts to be any value. Table 1 summarizes the result. We found a total of 1,588 system calls from `ntdll.dll` and `win32u.dll`. Since our analysis always finds a function that invokes a system call (with software interrupt), we can say that there are at least 1,588 system call stubs on Windows 10, which are potential instrumentation target for our fuzzer.

Due to the lack of documentation, about 100 out of 1,588 system calls are currently known. However, other system calls are not publicly known, and we do not even know their function prototypes. Without knowing the parameter types, we can only get the first-level

```
public NtAccessCheck
NtAccessCheck proc near
mov eax, 0
call sub_6A29044D
retn 20h
NtAccessCheck endp
```

```
sub_6A29044D proc near
mov edx, esp
sysenter
retn
sub_6A29044D endp
```

(a) Windows 10 x86

```
public _NTAccessCheck
mov eax, 0
mov edx, offset _Wow64SystemServiceCall
call edx
retn 20h
_NTAccessCheck endp
```

(b) Windows 10 x86-64 (WOW64)

Figure 2: Examples of Windows 10 syscall stubs.

pointer values for each system call parameter, and thus, it is not possible to obtain precise API model as in IMF on macOS.

To overcome this challenge, one may design a static analyzer that infers parameter types for each syscall stub we found. Indeed, this is the key intuition of our approach. There are publicly known Windows API functions (user-level), and those user-level functions often call kernel system API functions internally. Thus, our main aim here is to build a binary analyzer that infers parameter types of kernel API functions by propagating the type information obtained from the known user-level functions.

5 System Design

In this section, we present the overall design of our system for analyzing kernel binaries. Our target binaries include `ntdll.dll`, `kernelbase.dll`, `kernel32.dll`, `win32u.dll`, `gdi32.dll`, and `user32.dll`.

5.1 Analyzing Syscall Parameter Types

Our system first parses WinSDK header files to obtain type information of high-level user API functions. We leverage SAL (Source Annotation Language) [6] to understand the uses of the API parameters too. For example, SAL indicates which of the parameters are used as input or output. The main idea here is to correctly figure out the type information of known API functions, and then propagate such information to parameters used to call system calls.

Consider an example code snippet in Figure 3, where `API_X` is a known user-level API function, and `syscall_A` is a system call. We first figure out the types of the two parameters of `API_X`, and then propagate the information to function `f` to figure out the data types for the parameters of `syscall_A`. In this example, we know the system call takes in a `struct` as the first parameter, and one of the field of the `struct` has the `ACL` type. Similarly, we can easily see that the second parameter of the system call has a `HANDLE` pointer type.

To propagate the known data type information to system calls, we perform a flow-sensitive bottom-up style analysis from known API function parameters to system call parameters. To cope with loops, we unroll them by a factor of n, where n is a user-configurable parameter.

```
void API_X(HANDLE handle, ACL* acl) {
  ...
  f(handle, acl);
  ...
}

void f(HANDLE h, ACL acl) {
  Data *s = malloc(sizeof(Data));
  s->acl = acl;
  ...
  syscall_A(s, &h, 0);
}
```

Figure 3: Parameter type analysis.

5.2 Syscall Logging

After figuring out data types for system calls, we should log uses of the system calls by observing the real program executions. System call logs serve as a basis for generating a program for fuzzing the Windows kernel.

At a high-level, our syscall logger hooks Windows system call stubs in `ntdll.dll` and `win32u.dll` to log system call sequences with their parameter values. We execute a userland program with an input to log system calls because any userland program will eventually invoke a valid sequence of system calls to operate. Unlike IMF, however, we do not need to log the same program execution multiple times because we have already analyzed the parameter relationships during our static analysis phase.

The main aim of syscall logging is to provide a "seed" for kernel fuzzing. In other words, we take a valid sequence of system calls, and then mutate the system calls in such a way that the sequence does not violate syscall parameter relationships. If a return value of a syscall should be used in another syscall based on our inferred model, we connect them accordingly during the mutation phase.

5.3 Handling User-Mode Callbacks

Performing the above two steps is not enough to handle many system calls especially when they include user-mode callbacks. Several kernel functions take in a user-level callback function as input, which means they can run user-level functions during their execution. Oftentimes, this process is performed by fetching a function pointer from a callback function table, which stores every possible callback functions. If our fuzzers does not properly initialize the table, we may hit user-mode crashes during fuzzing.

Figure 4 describes the user-mode callback mechanism. The kernel calls the dispatch function, called `KiUserCallbackDispatcher`, to call the user-mode callback function, which is `__ClientThreadSetup`. When the callback function terminates, it should call the function `NtCallbackReturn`, which is defined in `ntdll.dll`, to jump back to the kernel code. The security impact of this user-mode callback mechanism is that both kernel and user code can reference the same memory object. In particular, when a kernel allocates a user object and

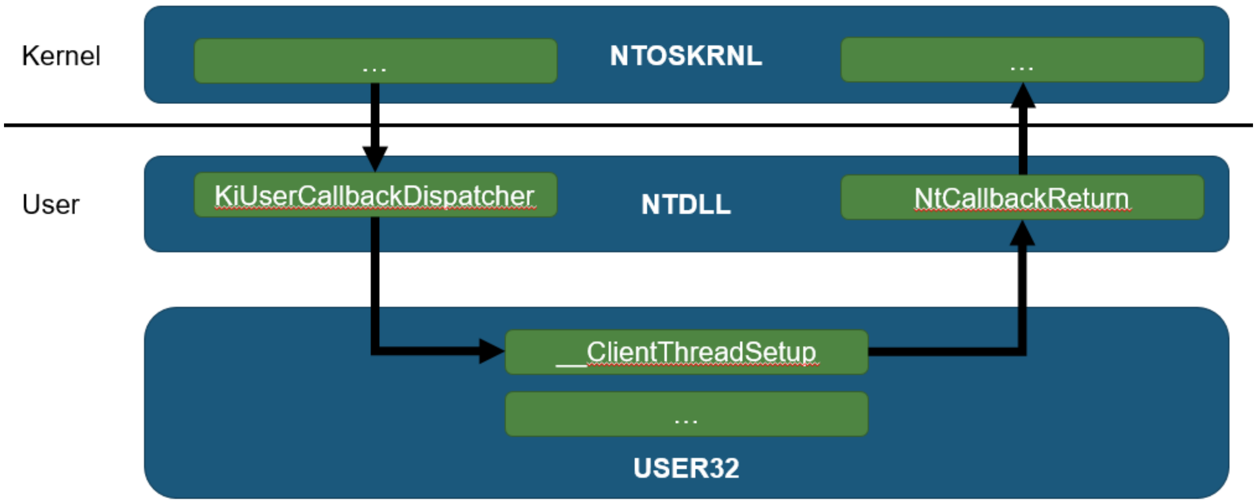

Figure 4: User-mode callback mechanism.

a callback function frees the object, then the kernel code may see a freed object, which leads to a use-after-free vulnerability.

To properly initialize the callback function table, we analyze our syscall traces to learn which functions are used as a callback, and we set the table to hole the list of observed callback functions.

5.4 Model Program Generation

Now that we have both (1) execution traces and (2) the type information of syscall parameters, the final step of our fuzzer is to create a model program in C to fuzz the Windows kernel. The program essentially contains various functions that invoke a system call while probabilistically mutating the parameter values. Since we already know the dependency relationships between syscall parameters, our model program should properly handle the dependency while generating the code. As an example, our model program looks as below.

```
ret = syscall_A(mut_int(1));
*buf = ...
syscall_B(mut_ptr(buf), mut_handle(ret));
```

There are two system calls A and B, and every parameter is wrapped by a mutation function, such as `mut_int` and `mut_ptr`, based on their inferred data types. For instance, when a parameter has an `int` type, then we randomly mutates the value of it to have a random integer value. However, we do *not always* mutate all the parameter values as doing so can break the dependency relationship between system calls, which will cause most of the system calls to fail in the end.

Note we do not mutate `HANDLE` type parameters as it can largely affect the success rate of system calls. This is a common design choice found in many kernel fuzzers for other OSes too, e.g., Syzkaller [23].

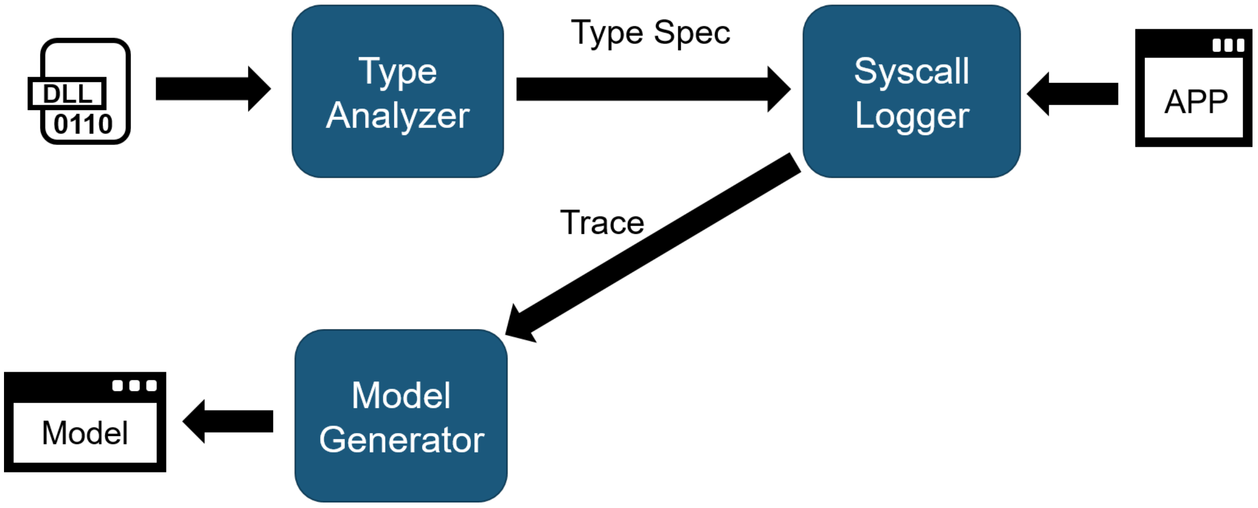

Figure 5: Our Fuzzer Architecture.

5.5 Implementation

Our prototype system consists of three main modules: type analyzer, syscall logger, and model generator. Figure 5 depicts the overall architecture of it. The type analyzer first analyzes Windows native binaries to infer system call parameter types. The syscall logger then instruments system call stubs to obtain system call traces while executing userland programs. Finally, the model generator use leverage the trace along with the type specification to create a model program for fuzzing the Windows kernel.

We implemented our system on our binary analysis framework [8], that we refer to B2R2. The system consists of 16K lines of F# code, and 1K lines of C++ code.

6 Evaluation

6.1 Evaluation Setup

We evaluated our system to check if it can find any realistic bugs on the Windows kernel. To run the system, we first took system call traces by running various user-land programs. Particularly, we manually crafted user inputs to run the six Windows programs: `Nodepad`, `Wordpad`, `SumatraPDF`, `Internet Explorer`, `Windows media player`, and `Chess`. The user inputs mainly consist of clicking mouse buttons and firing keyboard shortcuts in order to open a file. For example, we ran `Windows media player` and used an MP4 file as input. As a result, we obtained six different syscall traces.

6.2 Replaying Traces

To evaluate the precision of the model generated by our system, we first generated six different model programs from the six different traces. We then simply replay the trace by running the model program without mutating any parameter values and measured how many system calls were returning the same results compared to the original syscall traces.

	Notepad	Wordpad	SumatraPDF	IExplorer	WMPlayer	Chess
# of Succ. Calls	5,604	22,149	12,372	48,490	61,536	71,387
# of Calls	8,599	28,714	23,992	77,185	109,110	91,314
Success Rate	65.17%	77.14%	51.57%	62.82%	56.40%	78.18%

Table 2: The rate of successful system calls during replay.

Table 2 summarizes the results. The success rate was at least 51.5%, and 65% on average. This roughly means, our analysis can precisely model more than a half of the system calls. Note our goal is not on precisely modeling the system call specification, but rather on fuzzing system calls. We believe the current success rate of 65% is enough for effectively fuzzing the kernel to find vulnerabilities.

6.3 Bugs Found

Finally, we ran all the model programs for 24 hours each to find bugs resulting a kernel panic. We used the latest (at the time of writing) Windows 10 64-bit, version 1903. In total, we found 11 unique crashes based on stack traces, and we reported all of them to Microsoft. One of the reported bugs was assigned with a CVE number (CVE-2020-0792).

Listing 1 shows a simplified model program that our system automatically generated. To ease the representation we manually minimized the program. Although our experiment was limited to those six trace files we manually generated, it was able to effectively find many unique kernel panic bugs. We expect our fuzzer to be more lucrative in terms of bug finding, when we use more trace files.

Listing 1: Example model program that found a previously unknown kernel panic.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <windows.h>
#include <winnt.h>
#include <psapi.h>
#include <subauth.h>
#include "stub.h"

#define CALLBACK_TABLE_SIZE 0x400
#define TARG_ADDR ( 0xFFFFF78000000030 )
#define PADDING (0x40)

void * callback_table_nop[CALLBACK_TABLE_SIZE];

// Function body is definied in assembly code.
void set_callback_table(void* tablePtr);

void callback_nop(void) {
  ntdll_NtCallbackReturn(0, 0, 0);
}

void deactivate_callback(void) {
  set_callback_table(callback_table_nop);
}

void init_callbacks(){
  int i;
  // Initialize user-mode callback table.
  for(i = 0; i < CALLBACK_TABLE_SIZE; i++) {
    callback_table_nop[i] = callback_nop;
  }
  set_callback_table(callback_table_nop);

}

unsigned short register_win_msg(void) {
  unsigned short atom = 0;
  UNICODE_STRING str;
  wchar_t * name = L"myatom";
  size_t len = wcslen(name);

  str.Length = 1;
  str.MaximumLength = 1;
  str.Buffer = (wchar_t *) TARG_ADDR;
  atom = (unsigned short) win32u_NtUserRegisterWindowMessage((uint64_t) &str);
  printf("NtUserRegisterWindowMessage() = %hx\n", atom);
  return atom;
}

void get_atom_name(unsigned short atom) {
  UNICODE_STRING str;
  wchar_t buf[64];
  memset(buf, 0, sizeof(buf));
  str.Length = 0;
  str.MaximumLength = sizeof(buf);
  str.Buffer = buf;

  win32u_NtUserGetAtomName((uint64_t) atom, (uint64_t) &str);
  wprintf(L"NtUserGetAtomName(): Buffer = %ls\n", str.Buffer);
}

int main(void) {
  unsigned short atom;

  init_callbacks();
  atom = register_win_msg();
  get_atom_name(atom);

  return 0;
}
```

7 Conclusion and Future Plan

We have studied several challenges to apply model-based fuzzing technique on Windows kernel. One of the key challenges was to infer the type information of system calls, which are largely undocumented in the first place. We also have presented our prototype design for windows kernel fuzzer, and evaluated it on the latest Windows kernel, which results in 11 unique kernel panic bugs based on the stack trace. We plan to finalize our prototype implementation by employing more advanced mutation techniques, and we will submit a paper to CCS 2020, which is one of the most renowned conferences in security.

References

[1] The AV-test security report 2016/2017. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2016-2017.pdf.

[2] NT syscall information. https://undocumented.ntinternals.net/.

[3] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, 2008.

[4] J. brun. IOCTLbf. https://github.com/koutto/ioctlbf, 2017.

[5] K. Y. Chuan. Fuzzing the windows kernel. In *Proceedings of the Hack in the Box Security Conference*, 2016.

[6] M. Corporation. https://docs.microsoft.com/en-us/cpp/c-runtime-library/sal-annotations?view=vs-2019.

[7] H. Han and S. K. Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[8] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha. B2R2: Building an efficient front-end for binary analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[9] M. Jurczyk. NT syscall information by j00ru. http://j00ru.vexillium.org/syscalls/nt/64/.

[10] M. Jurczyk. Win32k syscall information by j00ru. http://j00ru.vexillium.org/syscalls/win32k/64/.

[11] M. Jurczyk. csrss_win32k_fuzzer. http://j00ru.vexillium.org/?p=1455, 2012.

[12] M. Jurczyk. Effective file format fuzzing. In *Proceedings of the Black Hat EU*, 2016.

[13] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 689–701, 2017.

[14] J. Loureiro and G. Geshev. Platform agnostic kernel fuzzing. In *Proceedings of DefCon*, 2016.

[15] Microsoft Corporation. Full windows api index. https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx.

[16] Microsoft Corporation. NtXxx routines. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/ntxxx-routines.

[17] MWR Labs. KernelFuzzer. https://github.com/mwrlabs/KernelFuzzer.

[18] NCC Group. DIBF. https://github.com/nccgroup/DIBF, 2014.

[19] Nils. Windows kernel fuzzing. In *T2 InfoSec Conference*, 2015.

[20] D. Oleksiuk. Ioctl fuzzer. https://github.com/Cr4sh/ioctlfuzzer, 2009.

[21] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the USENIX Security Symposium*, pages 729–743, 2018.

[22] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the USENIX Security Symposium*, pages 167–182, 2017.

[23] D. Vyukov. syzkaller. https://github.com/google/syzkaller.

[24] D. Weston and M. Miller. Windows 10 mitigation improvement. In *Proceedings of the Black Hat USA*, 2016.

[25] P. Yosifovich, A. Ionescu, M. Russinovich, and D. Solomon. *Windows Internals.* Microsoft Press, 7th edition, 2017.

[26] Zerodium. Zerodium payouts for desktops/server. https://zerodium.com/program.html.