A Framework for Diversifying Windows Native APIs to Tolerate Code Injection Attacks

Lynette Qu Nguyen Tufan Demir Jeff Rowe Francis Hsu Karl Levitt
University of California, Davis
{nguyenly,demirt,rowe,fhsu,levitt}@cs.ucdavis.edu

ABSTRACT

We present a framework to prevent code injection attacks in MS Windows using Native APIs in the operating system. By adopting the idea of diversity, this approach is implemented in a two-tier framework. The first tier permutes the Native API dispatch ID number so that only the Native API calls from legitimate sources are executed. The second tier provides an authentication process in case an attacker guesses the first-tier permutation order. The function call stack is back-traced to verify whether the original caller's return address resides within the legitimate process. The process is terminated and an alert is generated when an attack is suspected. Experiments indicate that our approach poses no significant overhead.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*invasive software, authentication, security kernels*; D.2.0 [**Software Engineering**]: General—*Protection mechanisms*

General Terms

Security, Diversity

Keywords

Diversity, Windows Native API, Code injection attacks

1. INTRODUCTION

A code injection attack is one of the most common techniques used by attackers to gain complete control of a victim machine. These attacks take advantage of software memory errors to inject malicious instructions into the victim's process memory. Due to the homogeneity of computer systems, a single vulnerability can result in a widespread compromise of hosts. This gives attackers the ability to launch Internet worms, maintain malicious botnet slaves, and conduct DDoS attacks.

Simply depositing and executing malicious instructions in process memory is, by itself, not sufficient to gain control of a victim. Reading or writing to the disk, connecting to the network, and many other critical operations require the use of Native API calls. In MS Windows, native API calls are the interface used by user processes to request services from the operating system. In Linux/Unix, system calls are trapped into the kernel through $int\ 0x80h$ after the system call number is pushed onto register eax. A Windows Native API call is similar; the Native API dispatch ID number is pushed onto register eax, the call is trapped into the kernel through an interrupt/fastcall, and the specified service is then provided by the kernel. The kernel serves requests it receives both from malicious injected instructions and from the legitimate user processes without distinctions.

We aim to diversify the MS Windows operating system such that even if one machine can be compromised, the same technique will not work on all others. This will prevent an attacker with a single exploit from obtaining widespread compromise. We choose the Windows operating system because the architecture flaws and the proprietary nature make it more vulnerable to attack. In addition, majority of PC users run Windows. We feel it is urgent to alleviate the damages on windows due to the large population that are affected. We have implemented a framework that wraps the application or service we want to protect. Our framework instruments the program binary at load time to add diversity to each software instance. The framework is an interface between the operating system's services and the software access that monitors all the Native API calls a program invokes. The term shellcode represents any byte code that is inserted into an exploit to accomplish a desired task.

2. BACKGROUND

Although Windows has many main sub-system DLLs such as $user32.dll$, $gdi32.dll$, $advapi32.dll$, $ntdll.dll$ and $kernel32.dll$, we focus on $ntdll.dll$ since it serves as the fundamental user-mode interface of the windows Native APIs. The actual implementations of the Native API calls reside in $ntoskrnl.exe$ which is in the kernel. Each Native API has a stub residing inside $ntdll.dll$. Figure 1a shows the Native API stub for $ntClose$. The first instruction loads register eax with the Native API's dispatch ID number. This dispatch ID is unique for each Native API and varies among different builds of the operating system. The second instruction loads register edx with the address of the fast call stub and the third instruction call into this stub.

All the native call stubs look almost identical with the

```
B8 19 00 00 00      mov     eax, 0x19
BA 00 03 FE 7F      mov     edx, 0x7FFE0300
FF 12               call    dword ptr[edx]
C2 04 00            ret     0x4
```

(a) *NtClose* Native API call

```
8B D4     mov     edx, esp
0F 34     mov     sysenter
C3        ret
```

(b) Fast call stub in Windows XP

Figure 1: Windows Native API call

one shown in Figure 1a. The fast call stub (Figure 1b) saves the stack pointer and transitions from the user mode to the kernel mode through *sysenter* instruction. When a Native API call service is requested, system service dispatcher $KiSystemService$ is called in the kernel. $KiSystemService$ use the Native API dispatch ID number stored in register eax as an index to lookup in the $System\ Service\ Dispatch\ Table(SSDT)$ which routes the call to its final destination. In the meantime, $KiSystemService$ also copies all the arguments contained in register edx onto the kernel stack. The current Windows XP system has 284 Native APIs exported by *ntdll.dll* with dispatch IDs ranging from 0 to 283.

3. DESIGN AND IMPLEMENTATION

3.1 Overview

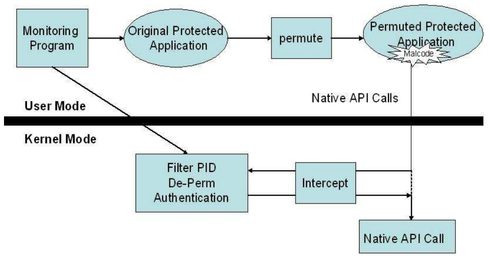


Figure 2: Overview of our framework

We implement our framework as a monitor that controls the load and execution of protected programs. Using a kernel device driver, our monitor process P executes the protected program as a child process C, examines and manipulates the memory space of process C, and intercepts Native API calls from process C. The monitor instruments the protected process C at link and load time by permuting the Native API dispatch ID number in memory. Then the kernel device driver intercepts all the Native API calls coming from the protected process. The Native API numbers are de-permuted for each intercepted Native API call before kernel serves the request. In this way, only Native API calls from the monitored process will function correctly. Unless an attacker can guess the permutation, Native API calls from injected instructions will malfunction due to an incorrect dispatch ID. Furthermore, in the unlikely event that an attacker guesses the permutation correctly, our framework does further authentication to ensure the API request is from the original code and not the newly injected code. When the authentication process sees API requests from invalid memory locales, the process is terminated and the user is alerted. Native API dispatch ID number permutation is achieved in two steps: when the application binary is loaded into the memory and when the Native API call is intercepted by the kernel device driver.

3.2 Permute at Load Time

Once the application binary is loaded into the memory, we suspend the thread and do a binary static analysis of *ntdll.dll* on disassembled instructions. We integrate and modify an existing open source disassembler [1] into our project. Once the Native API call instructions in the protected process are identified, the dispatch ID put in register eax is replaced with the permuted ID. Since injected malicious code is not present at load time, it will have different, likely original, dispatch IDs.

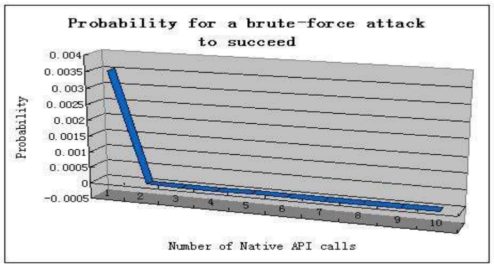
3.3 Intercept Native APIs

At run-time, the permuted dispatch IDs must be mapped back to the intended ID in the original API call for correct operation. To accomplish this, we implement a kernel device driver that "hooks" the $SSDT$. Traditionally, when a Native API call service is requested, the dispatch ID number stored in the eax register is indexed into the $SSDT$ to retrieve the address of the function that handles the API call. Our approach hooks the $SSDT$ to point to our code instead of the original address of the Native APIs. In this way, whenever an application calls into the kernel, the request is processed by the system service dispatcher which calls our function before the API call is routed through its actual address. This is a critical component of our kernel device driver since it performs the tasks of filtering PID (process ID), de-permutation and authentication. One issue with a kernel device driver is that it is system wide; all Native API calls from all processes in the system are hooked. This can crash the system since API calls from legitimate, unprotected processes do not get served correctly. We solve this by filtering on the PID of the protected process. If the Native API calls are from other legitimate processes other than the protected process, the request will be served as is. Otherwise, we perform de-permutation and route the API call request to the address in the corresponding call.

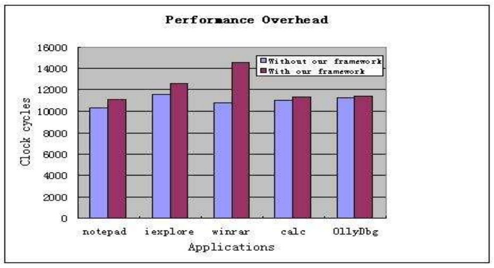
3.4 Authentication

Authentication is the second tier of the framework assuming the attacker has successfully guessed the permutation scheme and can access the Native API call services. To authenticate that the Native API calls are indeed coming from a legitimate requester, a back-trace of the function call stack is performed. The $x86$ architecture traditionally uses the *ebp* register to establish a stack frame. The return address is always located at offset $[ebp + 4]$ on the call stack. Thus, the return address is easily identified once *ebp* is located. By following the return address while back tracing the stack, we check whether the return address is within the protected process. If the traced return address is not within the application's address space, we suspect an attack and the process is terminated with a generated alert.

4. PROBABILISTIC EVALUATION



(a) Probability evaluation



(b) Performance evaluation

Figure 3: Probability and performance evaluations

Since our framework is based on the concept of diversity, it is crucial to determine the probability that an attacker can succeed by performing a brute-force attack. The probability for the attacker to successfully guess the correct Native API call number is $P(1\_call) = 1/284$. If the attacker succeeds on the first Native API call, the chance of success for the next is $P(2\_calls) = 1/284 * 1/283$. Thus, the probability for the attacker to succeed for n native API calls is:

$$P(n\_calls) = 1/284*1/283*\cdots*1/(284-n+1) = (284-n)!/284!$$

Figure 3a reflects the above equation. As figure 3a shows that once the number of native API calls exceeds 1, the probability for the attacker to succeed becomes very small. Our result is based on the assumption that the attacker has to make at least 2 Native API calls in order to control the machine. This probabilistic evaluation shows it is difficult for the attacker to break our randomization through a brute-force attack.

5. PERFORMANCE EVALUATION

We run our experiments on a 1.6 GHz Pentium M processor with 128 MB RAM. The implementation in this work is based on Windows XP with Service Pack 0. We start the experiment by measuring the cost of intercepting and authenticating an individual Native API call. Then we evaluate the performance overhead before and after applying our framework for selected applications. All the experiments are measured with the Pentium processor's $RDTSC$ (read time stamp out) instruction. To evaluate the execution time of the protected program and comparing the costs, we intercept only a limited number of Native API calls. We chose 5 commonly used applications to be our experiment targets. We perform 5 experiments on each of the 5 applications and each run intercepts 8 Native API calls. Figure 3b shows the clock cycle counts for each of the applications before and after our framework is applied. The figure indicates that the overhead is less than 3% for both *calc* and *OllyDbg* and less than 9% of overhead for both *notepad* and *iexplore*. The results vary based on the Native APIs called by each application. Our experiment on individual Native API call explains this variation. Thus, the overall performance is not significantly affected by our permutation and authentication scheme.

6. RELATED WORKS

Researches have been conducted regarding system diversity since the idea of randomization was proposed by Forrest et al. [2–5]. The ideas that authenticating system calls to prevent code injectioin attacks are proposed in [6, 7]. Rajagopalan et al. proposed a behaviour specification approach to verify a cryptographic message authentication code against a policy generated through augmented arguments [7]. Among the techniques proposed by Linn et al. [6] to counter code injection attacks, one approach is to add a new sectioin to ELF executable in order to distinguish syscalls invoked illegally from legally. This approach requires the modificaiton of the Linux kernel in order to incorporate with the information of the new section. Both of these two approaches are effective against code injection attacks. However, they are both implemented in Linux instead of Windows.

7. CONCLUSIONS

Most code-injection attacks must use Native API calls to do damage once the malicious instructions are deposited. We adopt the idea of diversity to permute the Native API dispatch ID numbers, causing the injected code to malfunction whenever a Native API call is issued. An attacker without knowledge of the permutation scheme can not easily bypass our monitor by guessing. By analysing and instrumenting the binary code of the protected process directly, our framework does not require access to source code and can be easily deployed without major changes to the system. The experimental results indicate minor run-time overhead. We believe the approach of automatically permuting and authenticating Native API calls is promising toward defeating code-injection attacks.

8. REFERENCES

[1] Bastard disassembler. http://bastard.sourceforge.net/.

[2] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. *USENIX Security Symposium*, 12(2):291–301, August 2003.

[3] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[4] J.Xu, Z.Kalbarczyk, and R.K.Iyer. Transparent runtime randomization for security. *Proceedings of 22nd Symposium on Reliable and Distributed Systems (SRDS)*, October 2003.

[5] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.

[6] C. Linn, M. Rajagopalan, S. Baker, C. Collberg, H. Hartman, and S. Debray. Protecting against unexpected system calls. *Proceedings of the USENIX Security*, pages 239–254, 2005.

[7] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated system calls. *International Conference on Dependable Systems and Networks(DSN '05)*, pages 358–367, 2005.