Shellzer: a tool for the dynamic analysis of malicious shellcode

Yanick Fratantonio[1], Christopher Kruegel[2], and Giovanni Vigna[2]

[1] Politecnico di Milano, Italy
yanick.fratantonio@mail.polimi.it
[2] University of California, Santa Barbara
{chris,vigna}@cs.ucsb.edu

Abstract. Shellcode is malicious binary code whose execution is triggered after the exploitation of a vulnerability. The automated analysis of malicious shellcode is a challenging task, since encryption and evasion techniques are often used. This paper introduces *Shellzer*, a novel dynamic shellcode analyzer that generates a complete list of the API functions called by the shellcode, and, in addition, returns the binaries retrieved at run-time by the shellcode. The tool is able to modify on-the-fly the arguments and the return values of certain API functions in order to simulate specific execution contexts and the availability of the external resources needed by the shellcode. This tool has been tested with over 24,000 real-world samples, extracted from both web-based drive-by-download attacks and malicious PDF documents. The results of the analysis show that *Shellzer* is able to successfully analyze 98% of the shellcode samples.

Keywords: Shellcode analysis, Binary instrumentation

1 Introduction

Malware, which is a generic term used to denote software that aims to compromise a computer, is the leading threat on the Internet. One of the primary methods used by the attackers to deliver malware is code injection.

In the case of web-based malware, the user is lured into visiting a malicious web-page. The JavaScript contained in that page tries to exploit a vulnerability in the browser. If it succeeds, the exploit triggers the execution of an arbitrary piece of code, often called *shellcode*. A shellcode is a small piece of code, whose goal is to compromise the machine that executes it. The size of shellcode samples is usually subject to some constraints, and, hence, their actions are commonly limited. Despite this, they play a fundamental role in the compromise of a host. The task of the shellcode is often to provide interactive access (through the invocation of a command line shell, from which the term "shellcode" is derived), or download and then execute additional malware.

In this paper, we introduce *Shellzer*, a tool for the dynamic analysis of malicious shellcode. In particular, we focus our attention on shellcode extracted from web-based malware and from malicious PDF documents. Given a shellcode in input, *Shellzer* analyzes it by instrumenting each instruction in the code. This is done to have complete control over the shellcode's execution, so that it is possible to detect the use of evasion techniques and to collect detailed information about the sample under analysis. Two different optimizations have also been introduced in order to make this approach feasible in terms of performance. Furthermore, in order to fulfill some specific conditions required for a correct analysis, the tool dynamically alters both the arguments and the return value of some API functions. By doing this, the tool is capable of observing the behavior of the shellcode during a real-attack scenario. This technique is used also to deal with the attempted malicious actions. In fact, since instrumentation is just a different form of execution, if no countermeasures were taken, the shellcode would be able to compromise the host that runs the analyzer.

As output, the tool returns an HTML report that contains the following information: a *complete* trace of the API calls (with their most significant arguments and their return values), the URLs from which external resources have been retrieved, and the evasion techniques used by the shellcode. Furthermore, the tool returns the additional binaries that have been downloaded at run-time. It is worth noting that even if the binary retrieved was originally encrypted, the tool will automatically return its decrypted version. This is a quite important feature of our work: indeed, having just the encrypted binary would be useless, since the decryption routine is implemented in the shellcode, and not in the binary itself. This key feature is useful also when dealing with shellcode samples extracted from malicious PDF documents. In these cases, the additional payload is contained in the PDF document itself, and *Shellzer* will be able to automatically return it, also if it was originally encrypted.

The tool has been evaluated by running it over 24,000 real-world samples, which had been previously extracted by Wepawet [5], an on-line service for detecting and analyzing different types of malware, including web-based malware and malicious PDF documents. The average time for a single analysis is 15 seconds, and only in the $\sim 2\%$ of the cases *Shellzer* has not been able to analyze the samples because of one of its limitations. During our discussion, we will provide an overview about the goals of this kind of shellcode, and we will describe some interesting samples we found during the analysis.

2 Issues to be addressed

In this section we discuss which are the common issues that make shellcode analysis difficult.

2.1 Additional resources have to be available

One of the challenges of shellcode analysis is that shellcode often requires that some additional resources (usually additional malware) have to be available. If the retrieval of such external resources fails, some samples silently quit, while others behave in an unexpected way (their execution usually crash). In both cases, this outcome is not desirable since our goal is to analyze the behavior of shellcode as if it were executed during a real-world attack.

2.2 A specific execution context is required

Many samples can be correctly analyzed only if they are executed in a specific execution context. A good example that shows why the context is an important aspect is the case of shellcode extracted from malicious PDF documents. Most of these samples make use of the `GetCommandLineA` API. This API returns a string that contains the command line that has been used to launch the program. The execution context is important here, because this kind of shellcode assumes that it is running inside an instance of Adobe Reader, and, therefore, it makes an assumption about how the string returned by that API is formatted. This point constitutes a big issue since if the shellcode is not executed in the appropriate execution context, the string returned by the `GetCommandLineA` API will be completely different from the one that shellcode expects, and, in the general case, this will cause some malfunctions (in the worst case, a crash).

2.3 Dealing with malicious behavior

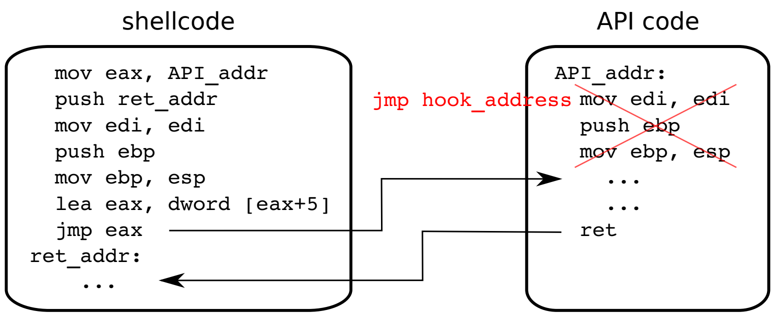
An important issue is related to the fact that the goal of the shellcode samples is to compromise the machine that executes it. This fact constitutes a problem for two reasons: the first is that our system actually executes the malicious shellcode, and hence there is a concrete possibility that the shellcode under analysis could be able to take control of the analyzer itself; the second one is related to the fact that we consider too expensive (in terms of performance) restoring the machine after the analysis of each sample.

2.4 Performance issues

Despite the fact that shellcode is usually few hundreds of bytes long, the number of instructions that are actually executed at run-time is in the order of millions. This is due to the fact that many loops are present, and some of them are executed thousands of times. This constitutes a big issue for our approach, since our system is based on single-instruction instrumentation, and, hence, the overhead introduced is directly proportional to the number of instructions executed at run-time.

2.5 Evasion Techniques

Malware authors often try to make shellcode analysis difficult. Specifically, the techniques that we are going to describe have the following goals: make the static analysis unfeasible, increase the difficulty of generating a complete trace of the API functions called, and mislead the analysis tools by performing some specific assembly-level tricks.



```
   mov eax, API_addr
   push ret_addr
   mov edi, edi
   push ebp
   mov ebp, esp
   lea eax, dword [eax+5]
   jmp eax
 ret_addr:
     ...
```

```
API_addr:
  ~~mov edi, edi~~
  ~~push ebp~~
  ~~mov ebp, esp~~
    ...
    ...
  ret
```

Fig. 1. `JUMP OVER THE HOOK` technique

Encryption Many of the samples we have analyzed were encrypted. Mainly two techniques are used. In the first case, the encryption is done by simply xoring all the bytes of the shellcode with a one-byte key. The second technique is also based on the xor operation, but the shellcode in encrypted in blocks of four bytes with a four-byte key. Moreover, in these cases, the key is altered for each iteration. When the second variant is used, encryption makes the analysis tools based on static approaches unable to extract any useful information.

Uncommon API functions Since the Windows DLLs export a huge numbers of API functions, many malware analyzers do not monitor the calls to *all* the API functions, but only the calls to a subset of them (i.e., the security relevant ones). Malware authors leverage this bias, and they use certain uncommon API functions that, with a high probability, are not monitored, despite the fact that this leads to an increase in the shellcode's complexity. For example, in order to run additional executables, the `WinExec` API is commonly used. Instead, some samples use the `CreateProcessInternalA` API, an undocumented function that takes twelve input arguments.

Assembly-level tricks We will now describe some low-level techniques that from the behavioral point of view do not add any contributions, but are used by shellcode authors to mislead analysis tools.

INDIRECT API CALL This technique allows the shellcode to call an API function **A**, by jumping into the code of a different API function **B**, with the aim of misleading some analysis tools. This is achieved in a simple way. When the shellcode has to call the API function **A**, it jumps to a specific point of the API function **B**, so that, after few assembly instructions, the API function **B** will internally call the API function **A**.

JUMP OVER THE HOOK This technique is very powerful and it constitutes a big issue for many analysis tools, since it makes it difficult to generate a *complete* trace of the called API functions. Specifically, it affects the monitoring tools that track the API calls by modifying the first bytes of each API, in order to install a hook on each of them. This technique comes into play when the shellcode calls an API function, and its goal is to jump over the installed hook. This technique simply consists in jumping to the fifth byte of the API, instead of jumping to the first one (as it happens in the normal case). Therefore, in the case a hook is present, it will be bypassed and the call to that API function will not be traced. This technique skips the first five bytes because it is the common size of a hook. Figure 1 shows this technique in action. Before calling the API function, the shellcode executes those assembly instructions that were at the beginning of the target function, so that the net effect of calling the API function is not altered.

RETURN ORIENTED PROGRAMMING Occasionally, shellcode uses a simple form of return-oriented programming. Since this topic has been widely discussed in literature, we will just discuss an example in order to show the low complexity of the samples we found.

Listing 1.1. `RETURN ORIENTED PROGRAMMING` assembly

```
push arg2_n
        ...
push arg2_0
push return_address
push arg1_n
        ...
push arg1_0
push API_2_address
jmp  API_1_address
```

Listing 1.1 shows a small excerpt where the shellcode uses this technique to call two different API functions in a row, without the need to come back to the shellcode's code between the execution of the first function and the second one.

3 Overview of the system

In this section, we provide an overview of *Shellzer*, the shellcode analyzer we have designed and implemented. From a high-level point of view, *Shellzer* takes as input the shellcode that has to be analyzed, and it generates an HTML report that includes the following information: the trace of *all* the API functions called (with their arguments and return values), the DLLs loaded, the URLs contacted, and the evasion techniques that have been used by the sample under analysis. In addition to the binary, the analyzer takes as input all the additional resources that are required at run-time. For example, if a shellcode has been extracted from a malicious PDF, the PDF itself is usually needed in order to correctly analyze the sample, and, for this reason, it has to be passed as input to our tool. Moreover, it often happens that the shellcode tries to retrieve and run external executables (presumably malware): if this is the case, these additional binaries are also returned as output. It is worth noting that, even if the binary was originally encrypted, its decrypted version will be returned, since the tool extracts the binary image of the additional component after it has been decrypted by the shellcode.

3.1 Architecture

Shellzer dynamically analyzes shellcode samples by instrumenting their execution at a single-instruction level granularity. The instrumentation is performed by using PyDbg [7], a Python Win32 debugging abstraction class. In particular, the core of the analysis is performed in the `EXCEPTION_SINGLE_STEP`'s handler that it is called between the execution of each assembly instruction.

Our goal was to have complete control over the shellcode's execution, as if we were using an approach based on emulation. The advantage in using such a technique is that we can dynamically decide if it is necessary to single-step through the code or not, so that the overhead caused by the instrumentation is introduced only when it is strictly required. We will now discuss the three main components of our system.

Advanced tracing technique. One of the most important aspect that has to be monitored is related to the Windows API functions that are called by the shellcode. In particular, it is important to be able to retrieve the names, the argument values, and return value of *all* the API functions that are invoked. Unfortunately, in the general case, this task cannot be accomplished in a straightforward way, since the shellcode might use evasion techniques (e.g., those presented in Section 2.5). For this reason, we had to develop a quite complex tracing technique through which we are able to automatically handle all the possible situations and all the evasion techniques known to us. We describe the details of this technique in Section 4.1.

Dynamic interaction. By using this kind of approach, we are able to inject custom pieces of code at any moment during the execution of the shellcode that is instrumented. This is a very powerful feature, since the injected code is executed in the same execution context of the shellcode under analysis: this means that it is possible to dynamically read and write the process memory, read and write the values of the CPU's register, and so on. Our tool exploits this capability to handle three of the major issues we presented in the previous section.

The first issue comes from the fact that the shellcode might try to retrieve additional resources and, if they are not available, the shellcode might behave in an unexpected way. Therefore, we dynamically simulate that the required resources are available by properly altering the return values of some specific Windows API functions.

The second issue we addressed is related to the fact that some shellcode needs to be executed in a very specific execution context. In order to solve this problem, we simulate that the whole analysis is performed within the required execution context. Also in this case, this is obtained by modifying, at run-time, the content of some specific memory regions and the return values of certain Windows API functions.

Finally, since an instrumented execution is still an execution, we have to handle the malicious nature of the shellcode samples we analyze: if no coun-

termeasures are taken, the shellcode under analysis could easily compromise the machine that runs the tool, and we would be forced to restore the host after each single analysis. This issue is resolved thanks to the fact that we have complete control over the shellcode's execution. Therefore, we modify the argument values of some security-relevant Windows API functions. We will discuss in more depth these modifications in Section 4.2.
Performance speed-up. Instrumenting the whole shellcode's execution with a single-instruction granularity bears a significant performance overhead. For this reason, we implemented two optimizations that allow for the disabling of the single-step mode when it is not necessary. The first aims to disable the single-step mode during the execution of API functions, while the second is related to the loops that are often present in shellcode: once the loop's body has been analyzed the first time, the single-step instrumentation is often no longer needed for the other iterations. The details related to these optimizations are discussed in Section 4.3.

4 Analysis process

We will now describe how the analysis is performed. Our tool takes as input a shellcode. Since PyDbg does not allow the instrumentation of code fragments, we first build an executable starting from the shellcode. This is done by dynamically writing a C program, whose `main()` just executes the shellcode's binary. After the executable is created and loaded by the debugger, we set a breakpoint on the shellcode's entry point (i.e., its first byte), whose position in memory is statically known, and we then run the executable. When the execution reaches the first byte of the shellcode, the breakpoint is triggered and the remaining initialization steps are performed.

Firstly, the tool collects some information related to the DLLs and the API functions whose usage has to be monitored. In particular, it is determined which DLLs have been loaded and at which addresses in memory they have been mapped. The system then retrieves from a configuration file which API functions are exported by those DLLs, their addresses in memory, and the arguments whose values have to be reported in the output trace. For each argument, the following information is gathered: the *name*, used as a label to identify it; the *type*, necessary in order to properly interpret the bytes read from memory; the *input/output* flag, that indicates if the memory has to be read before or after the API function's execution; and the *offset*, which allows one to determine the argument's position in memory. Specifically, the *offset* indicates the argument's position with respect to the value assumed by the stack pointer just before the API function's first instruction is executed.

At this point, some specific handlers are registered for the most common Windows exceptions (e.g., `EXCEPTION_ACCESS_VIOLATION`) that could be raised during the analysis. Then, a handler for the `EXCEPTION_SINGLE_STEP` exception is registered. This handler plays a fundamental role in our system, since it is the place where the analysis happens. Indeed, that function is executed between each shellcode's assembly instructions. This is achieved by setting the `TF` flag of the `EFLAGS` register before the execution of each instruction: in this way, after the current instruction will be executed, the processor will execute the `int 0x1` interrupt, the `EXCEPTION_SINGLE_STEP` exception will be raised by Windows, and our handler will be then called.

What we have described so far are the operations performed during the initialization phase. In the remaining of this section, we discuss three important aspects of our system, namely, how *Shellzer* traces API functions, how it handles the correct and safe execution of API functions, and how it optimizes the performance of the shellcode analysis.

4.1 API calls detection and tracing

We now describe how *Shellzer* detects and traces the API functions called by the shellcode. The operations discussed here are performed before the execution of each assembly instruction. First of all, the value of the program counter (`PC`) is retrieved. We then determine if the `PC` points to a memory region where a DLL has been mapped: if this is the case, the DLL's name is retrieved. At this point, we compare the `PC`'s value with the starting address of all the API functions exported by that DLL. If a match is found, we are able to determine which specific API has been called, and we can proceed in reading the arguments' values, whose position can be calculated by adding their *offset* attribute to the current value of the stack pointer (`SP`). Once this information has been collected, the API function called by the shellcode can be executed. When the execution returns (we will see how this can be determined in a reliable way), the API function's return value is retrieved by simply reading the value stored in the `eax` register. The last operation performed before continuing with the analysis consists in the retrieval of the API's output arguments, that can be now read.

The procedure described so far works only if the shellcode called the API function in a conventional way; however, if the evasion technique we previously named `JUMP OVER THE HOOK` is used, additional operations are required. Indeed, the tool detects that an API function has been called only when the execution reaches the API's code, and, in the case such evasion technique is used, this will occur only after the first three API's instructions (which represent the first five bytes of the function) have already been executed by the shellcode (details have been explained in Section 2.5). This constitutes an issue both in determining which API function has been called and in properly retrieving the arguments' values. In fact:

- the current value of the `PC` will no longer match the starting address of any API function, since it will point to the fourth API function instruction and not to the first one;
- the value of the *offset* attribute of each argument, through which the argument's position is found, is relative to the value assumed by the `SP` just before the execution of the first API's instruction, which, in the general case, will be different from the current value of the `SP` register.

For what concerns the identification of the API called, the problem is solved by identifying which is the API (if any) that satisfies the following expression:

`API's starting address` $\leq$ `PC` $\leq$ `API's starting address + 5`.

If such API is found, we assume that the shellcode wanted to call that specific API and that wanted to avoid a possibly installed hook.

To correctly retrieve the argument values, we calculate the value that would be assumed by the `SP` just before the execution of the first API's instruction. This is done by considering the current `SP` and by modifying its value, taking into account the API's instructions already executed. For example, if between the first instruction and the fourth instruction, the shellcode has executed a `push ebp` instruction, it means that the `SP` has been altered by subtracting 4: so, in order to retrieve the `SP`'s original value, we add 4 to the current `SP`.

Note that, it is always possible that the shellcode jumps in the DLL's code, but we are not able to identify which API has been called: this can happen, for example, if the shellcode uses the technique we named `INDIRECT API CALL`, described in Section 2.5, or if the shellcode jumps in the code of an API function after its fifth byte (this circumstance has never occurred during the evaluation phase). In this case, the tool simply continues with the single-step analysis. At a certain point, either the execution will come back to the shellcode or an API will be called.

4.2 API handling

Our system does not perform only passive monitoring of the shellcode's behavior, since it would not be sufficient to handle the issues discussed in Section 2. More precisely, the tool allows one to execute user-specified functions just before and after the execution of a generic API function. By doing this, the analyzer is able to read and write the memory and alter the register's values. As an additional feature, we designed the system in a way that the shellcode can execute a specific API function only if that API function has been explicitly labeled to be *safe*. An API is considered to be *safe* in two cases: first, if its execution cannot lead to any bad consequences; second, if its execution could be problematic, but actually it is not, thanks to the additional operations performed before and after executing it. In this way, if the shellcode suddenly start to use a new API that is not currently handled (i.e., it is not *safe*), the tool will raise an exception and the anomaly occurred is signaled in the report generated. We now describe in detail the additional operations performed and which issues they address.

Resource availability simulation A shellcode sample might try to retrieve additional resources. If it is not able to do that, it might behave in an unexpected way. Unfortunately, the retrieval of such external resources is not always possible. For example, a common reason is that the URL contacted is no longer operational.

In order to address this issue, we inject our custom code before and after the execution of the API functions that deal with the downloading of the additional resources. For example, we do so for the `URLDownloadToFileA` API function. This API takes as input the URL from which the resource has to be fetched and the path where the file downloaded has to be stored on the file system. As output, it returns 0 if the file has been successfully downloaded, while it returns 1 if an error occurred. After this API function has been executed, the tool automatically checks its return value, which is stored in the `eax` register. If the value indicates that something went wrong, these two operations are performed: we change the value stored in the `eax` register to 0, and we create a file where the shellcode expects to find the resource downloaded. What we found is that these two operations are sufficient to simulate that the resource has been correctly downloaded, since the shellcode only checks the return value and not the content of the file retrieved. Of course, we need to do the same also when similar API functions are called, such as the `InternetReadFile` API function.

Environment simulation Some shellcode must run within a specific execution context. This is especially true when dealing with samples extracted from malicious PDF documents. These samples usually retrieve the additional malware not from the web, as it commonly happens, but from the PDF document itself. In order to do this, they first have to access the malicious document. In a real attack scenario, the shellcode is executed in the same execution context of the Adobe Reader instance that opened the malicious PDF document. In order to better explain the motivations behind this observation, we briefly summarize what happens from the moment the user opens the malicious PDF document, up to the shellcode's execution.

When the user opens the PDF document, Adobe Reader is executed and it reads and interprets the document's content. Moreover, if the document contains a piece of JavaScript, the reader executes it. What distinguishes a malicious PDF document from a benign one, is that the JavaScript tries to exploit a vulnerability in the reader. If it succeeds, it will able to trigger the execution of a piece of arbitrary code (in our case, the shellcode) that will be executed in the same execution context of the PDF reader.

We now present two examples of environment-aware shellcode. The first one is based on the `GetCommandLineA` API function. This API returns a string that contains the command line used to launch the program. If this API is called inside an instance of Adobe Reader, the output will be something similar to `"c:\Programs\adobe.exe" "c:\Documents\document.pdf"`, where the first is the complete path to the Adobe Reader's executable, and the second one is the complete path to the PDF file opened. This returned string is important, because the shellcode uses it to locate and then access the malicious PDF document.

The problem here is that the shellcode analysis is not performed within a real instance of Adobe Reader. In order to address this issue, we use the same mechanism we previously described: when the `GetCommandLineA`'s execution is terminated, the tool automatically modifies the returned string. Of course, this is done coherently with respect to what the shellcode expects, so that the PDF document can be correctly retrieved.

In the second example, when the shellcode's execution is triggered by the JavaScript code, Adobe Reader should have already opened the malicious PDF and, hence, a file handler associated with that document should be available within the current execution context. From a practical point of view, this technique aims to reuse the file handler associated with the PDF document, previously obtained by the Adobe Reader instance. In this case, the shellcode has just to determine which is the correct file handler: this can be done by properly using the `GetFileSize` API function. This API function takes as input a file handler and returns the size of the file associated with it. What shellcode usually do is a sort of brute-forcing: they repeatedly call the `GetFileSize` API function by passing as input all the different possible handlers. The shellcode will be able to determine which is the correct handler by comparing the API's return value, with the size of the malicious PDF document (that is known to the shellcode): when a match is found and hence the correct handler has been determined, the shellcode can then access the malicious PDF document and extract the additional malware.

In order to address this issue, a piece of code that simply opens the PDF document is executed before starting the shellcode's analysis. In this way, a file handler associated with the PDF document is available within the current context execution, and the shellcode will be able to access the malicious PDF document.

Security measures There are some API functions that, if called by a shellcode in a proper way, can compromise the machine that runs the analyzer. We now discuss two significant examples, and show how we have addressed these issues.

The first one is related to the `WinExec` API function. Through this API function (and a few similar ones), the shellcode can run an arbitrary executable. The problem here is that malicious shellcode uses this API function to run malware. Therefore, we dynamically alter the API's argument that indicates the path to the executable that has to be run. In this way, the shellcode, instead of running malware, will execute a fake program that simply sleeps for a while. Therefore, after this substitution, the code of the `WinExec` API can be safely executed, without any negative consequences.

The second example is related to the fact that shellcode could create files in arbitrary places in the file system. Of course, this can be a problem in the long run, since our tool needs to be able to analyze thousands of samples. We consider the `CreateFile` API function as an example for our discussion. This API function takes as input, among other arguments, the path of the file that has to be created. Also in this case, as a countermeasure, we modify at run-time the value of that specific argument: in this way, instead of creating a new file in an arbitrary directory, the shellcode will create a file in a temporary directory of our choosing, which is emptied after each analysis. During the analysis, we also maintain a mapping between the real file paths (the ones specified by the shellcode) and the temporary file paths (the ones specified by our tool). By doing this, if the shellcode requires to access a file that has been previously created, we are able to modify the file path consistently. Finally, it is worth noting that this mechanism comes into play when all those API functions that involve a file creation are called, including the `URLDownloadToFileA` API function we previously discussed.

4.3 Performance Improvements

Instrumenting the execution of all the assembly instructions is not feasible, because the time required for a single analysis would be too long (several minutes per sample). This is because the number of instructions executed at run-time is often in the order of millions, even if the shellcode samples are usually not bigger than a few hundred bytes.

In order to address this problem, we introduced two different optimizations, so that the shellcode's execution is instrumented only when it is really necessary.

Skipping the API's code Once an API is called, and the specific handlers (if any) have been executed, we disable the single-step mode (i.e., we do not set the `TF` bit of the `EFLAGS` register). By doing this the API's execution will not be instrumented. Of course, when the API function returns, we need to set again the `TF` bit in order to resume the normal analysis.

The difficult point here is to detect, in a reliable way, when the API function's execution is finished. To do this, we exploit the following observation: when an API is called, the caller has to push onto the stack the *return address*, i.e., where the execution has to jump once the API function returns. This implies that, just before the execution of the API function's first instruction, the *return address* will be the value on the top of the stack. When the tool detects that an API has been called, it performs the following operations:

1. it retrieves the *return address* pushed by the shellcode (that is the value on the top of the stack);
2. it sets a breakpoint on the *return address*;
3. it clears the `TF` bit, and it resumes the execution.

When the API's code will be completely executed, the execution will jump to the *return address*, and the breakpoint will be hit: at this point, the tool sets again the `TF` bit and the normal analysis is resumed.

Unfortunately, these operations are sufficient only if the shellcode *normally* calls the API. If a technique like the one we named `JUMP OVER THE HOOK` is used, the value assumed by the `SP` just before the execution of the first API's instruction has to be determined (we previously described how this is done): once that value is known, the *return address* can be retrieved in a straightforward way.

Furthermore, we need to resolve another problem: if the shellcode uses the technique we named `INDIRECT API CALL` or even simple cases of *return oriented programming*, it is possible that the breakpoint is hit and the API's execution is not finished yet. In order to understand if the API function has really returned, we exploit the following information: the value assumed by the stack pointer when the API's execution has really ended (`expected_SP`) can be calculated starting from the current value of the `SP`, by using the following expression:

```
expected_SP = current_SP + (#args + 1) * 4,
```

where `#args` represents the number of arguments to the specific API function called. Specifically, when the breakpoint is hit, the tool determines that the execution really came back from the API function's code to the caller, only if the `SP` matches the `expected_SP`. If this is not the case, it means that the API's execution has not terminated yet, and the breakpoint is set again (on the same *return address*). Then, the analysis can continue with the single-step mode disabled, until the API's execution has really ended.

Loop handler algorithm When analyzing shellcode, the number of the instructions executed at run-time is in the order of millions, despite the fact that the code is usually constituted of a few hundred bytes. This is because the shellcode often contains some loops whose body is executed thousands of times: in particular, loops are used for implementing the decryption routines and the techniques used to resolve the API addresses, as fully explained in [19]. Since single-step instrumenting all the iterations of each loop significantly hurts the performance, we designed a loop handler algorithm. Its goal is to provide a mechanism to disable the single-step instrumentation while the loop's body is repeatedly executed, and to re-enable it once the iterations are ended.
Overview. The algorithm is structured as follows. The first step is to determine if the execution is in a loop. If this is the case, the loop's body is analyzed in order to determine which are the exit points, i.e., the set of addresses such that at least one of them has to be reached once the loop's execution is terminated. Once this is done, a hardware breakpoint is set on each of them, the single-step mode is disabled, and the shellcode's execution is resumed. When the loop's iterations end, one of the breakpoint will be hit, and the tool will be able to re-enable the single-step mode in order to continue with the normal analysis. Furthermore, the tool maintains a structure that maps the first address of the loop's body with the set of the associated exit points. This is done for optimization purposes (if the execution reaches again the loop's starting address, the loop's body does not have to be re-analyzed), and, as we will see, for properly handling nested loops.
Loop detection. In order to detect when the execution is in a loop, the tool maintains an ordered list of the addresses of the instructions executed. In order to build this list, the value of the program counter is retrieved and appended to the list, before the execution of each instruction. Then, the list is walked backwards in order to find if the current instruction has been already executed in the past. If this is the case, the execution is in a loop, and the loop's body will be constituted by the list's entries between the two occurrences of the current instruction's address (i.e., the one that triggered the loop's detection). Actually, our algorithm is not currently able to handle loops that contain instructions like `jmp eax`, since the exit points cannot be statically determined if such instructions are present. What we do to handle this problem is to clear the list each time this kind of instructions has to be executed. In this way, the tool will lose the information related to which instructions have been executed in the past, and this will prevent the detection of a loop that includes such dynamic instructions.

Listing 1.2. `calc_exit_points()`

```
def calc_exit_points(loop_body, loops_detected):
    exit_points = set()
    candidates = set()
    for address in loop_body:
        if address in loops_detected.keys():
            candidates.add(loops_detected[address])
            continue
        ins = disassemble(address)
        if ins in branches:
            taken, not_taken = get_dest_addresses(ins)
            candidates.add(taken)
            if not_taken not None:
                candidates.add(not_taken)
    for candidate in candidates:
        if candidate not in loop_body:
            exit_points.add(candidate)
    loops_detected[loop_body[0]] = exit_points
    return exit_points
```

Similarly, the list is cleared also when an API function is called, so that, if a loop contains a call to an API function, its execution will be always single-step instrumented.

Determining the exit points. Listing 1.2 shows how the exit points are determined starting from `loop_body` (that is the list of the addresses that constitutes the loop's body), and from `loops_detected` (that is a map between the starting address and the associated exit points of the loops previously detected). What we do is to first build a set of *candidates*. For each instruction that constitutes the loop's body, we check if it is the starting address of an already-detected loop: if this is the case, the exit points associated with it are added to the *candidates* set. This is done because only the instructions whose execution has been single-step instrumented will appear in the `loop_body` variable. Therefore, in the case of nested loops whose execution is skipped thanks to the loop handler, only their starting addresses will be included in the `loop_body` list. For this reason, we have to consult the `loops_detected` variable in order to take into account the contributions (in terms of exit points) of the nested loops.

Then, we check if the current instruction is a branch instruction: if this is the case, we determine its two target addresses (or just one, if the instruction is an unconditional jump), and we add them to the *candidates* set.

Starting from this intermediate set, we are now ready to build the *exit points* set: this set will be populated by all those addresses contained in the *candidates* set that point outside the loop's body.

Final steps. At this point, a hardware breakpoint is set on each of the exit points, the single-step mode is disabled and the shellcode's execution is resumed.

When the loop's iterations will be terminated, one of the breakpoints will be hit, and the tool will be able to resume the single-step instrumentation.

The main reason for using hardware breakpoints is that the shellcode can easily detect the usage of software breakpoints by calculating a CRC (since these breakpoints need to modify the memory), while the presence of hardware breakpoints is much more difficult to be detected. But the usage of hardware breakpoints carries a big disadvantage: due to a processor's limit, only four of them can be used at the same time. This implies that our loop detector algorithm will not be able to handle more than four exit points: if a situation like this occurs, the tool will continue the single-step analysis, instead of trying to skip the loop. This circumstance has never occurred during the evaluation phase.

4.4 Evasion possibilities

We now make some considerations about how shellcode could detect our tool and evade our analysis. We also discuss some possible countermeasures.

Firstly, shellcode could determine that its execution is single-stepped. This can be done by checking if the `TF` bit of the `EFLAGS` register is set. The `EFLAGS` can be read by executing the `pushfd` instruction, that pushes the register's content onto the stack. As a countermeasure, when the shellcode executes that specific instruction, we modify the value pushed onto the stack by overwriting the `TF` bit with 0, so that the single-step mode will always appear to be disabled.

The shellcode could detect the usage of hardware breakpoints. One way would consist in reading the content of the Debug registers, but this cannot be done by processes executing with non-kernel privileges. Another way would be to install a custom exception handler and then execute an instruction that voluntarily raises an exception. By properly reading the Context structure, that is passed as a parameter to the exception handler, the content of the Debug registers can be read (this procedure is fully explained in [18]). As a countermeasure, the tool could act in the following way. When an exception is raised, the tool checks if a custom exception handler has been installed: if this is the case, the execution is redirected to the handler's code, and the Context structure is altered by properly overwriting the Debug registers' content. Actually, the technique that consists in installing a custom exception handler and then raising an exception could be used not only to detect breakpoints, but also to mislead tools based on debugging (like ours). But since the presence of custom exception handlers can be always determined, the tool could properly redirect the execution to the handler's code. More in general, there are many other anti-debugging techniques (like the ones described in [17]) that could affect our tool. But since *Shellzer* has the *complete* control over the shellcode's execution, it should be always possible to implement specific countermeasures.

Another way to mislead our tool could be to modify the API function's code. However, these modifications can be easily detected by a trivial integrity check and, if the shellcode calls an API that has been previously altered, the tool simply executes its code with the single-step mode enabled.

Finally, a problem that affects *Shellzer*, as well as most of the tools based on a dynamic approach, lies in the fact that only one execution path is examined during a specific analysis run. Therefore, if a shellcode expresses its malicious behavior non-deterministically, the report generated might be incomplete. To address this issue, we are planning to add to the output report an estimation of the code coverage, that should give an idea about how many shellcode's instructions have not been executed, and hence how much of the functionality might have been remained hidden during the analysis.

5 Evaluation

We will now describe how *Shellzer* performs at analyzing real-world samples. As a dataset, we used the Wepawet's shellcode database. Wepawet [5, 10] is an online service for detecting and analyzing different types of malware (web-based malware, malicious PDF documents, and others), and several thousands of resources have been submitted during the past few years. Its shellcode database has been filled with all the shellcode detected during the analysis of those submissions. Specifically, the detection is performed by applying several heuristics on strings longer than a certain threshold that contain non-printable characters. We now present the results we obtained. In the following, we will also discuss some interesting samples we found during the analysis.

5.1 Tool Evaluation

Our dataset is constituted by 29,873 samples. Unfortunately, it turned out that not all of them were actually valid shellcode. In particular, we found that 5659 entries were not valid shellcode: many of them are pieces of NOP-sleds that Wepawet wrongly considered to be complete shellcode. We analyzed the remaining 24,214 samples by setting a timeout of 60 seconds, after which the process is forced to be terminated. Table 1 summaries the analysis results. *Shellzer* has been able to *fully* analyze 20,306 (84%) of them. With the term *fully*, we mean that three requirements are satisfied: it has been possible to analyze the shellcode from the beginning to the end; no exceptions were raised during the analysis; no external resources required for the correct analysis were missing.

The average time needed for a single analysis is about 15 seconds, but it can greatly vary from case to case. If the shellcode simply downloads and executes an external resource, the analysis usually lasts about 5 seconds. However, if such external resource is not available, the time needed increases, since some Windows API functions (e.g., the `URLDownloadToFileA` API function), wait ~ 5 seconds before the execution returns to the shellcode (a sort of internal timeout is implemented). Furthermore, the time required is higher than usual also when the shellcode downloads a big file by calling thousands of times the `InternetReadFile` API function to fetch only few bytes at a time.

As we said, we obtained such great results in terms of performance, thanks to the two optimizations introduced. Table 2 helps to understand their importance.

Table 1. Analysis results

Label	Description	Number
A	Not valid	5659
B	Correctly analyzed	20306
C	Missing resources	2580
D	Corrupted samples	896
E	Timeout expired	400
F	Not *safe* API usage	32

Table 2. Optimizations' impact

Optimizations enabled	# of single-stepped assembly instructions
None	37232470
API Skipping	1173338
API Skipping Loop Handler	639

Specifically, the table shows the number of instructions that had to be single-step instrumented during the analysis of a simple shellcode, depending on the enabled optimizations: if both are used, the number of the single-stepped instructions is tremendously reduced.

With reference to Table 1, we now discuss which are the causes that prevent a complete analysis of the remaining 3908 samples.

C - Missing resources. Some samples download a binary into a buffer in memory, and then the execution jumps there. In 2580 cases (10%), the external binary was no longer available, and hence the analysis had to be stopped.

D - Corrupted samples. During the analysis of 896 samples (4%), an unexpected exception has been raised. To the best of our knowledge, these samples are somehow corrupted.

E - Time constraint. In 400 cases (2%), 60 seconds have not been sufficient in order to perform a complete analysis. Usually, this is caused by the fact that some samples implement loops in a way that our algorithm cannot handle them: in those cases, *Shellzer* has to single-step instrument the execution of all their iterations, and this causes a huge performance overhead. In these cases, the time required for the analysis usually varies between 4 and 5 minutes.

F - Not *safe* API usage. If, during the analysis, a shellcode sample calls an API function that is not considered to be *safe* (i.e., an API function that is not handled), the execution is interrupted as a safety measure, and hence the analysis is stopped. It is interesting how this happened only in 32 cases ($\sim 0.1\%$), even if the tool considers *safe* only 74 API functions. The API functions that are currently not handled by *Shellzer* are the ones exported by the `ws2_32` DLL (needed to perform remote connections), the `advapi32` DLL (useful to interact with the Windows Registry), and some others low-level API functions, such as `VirtualQueryEx`, `CreateRemoteThread`, and `SetUnhandledException`.

In conclusion, the analysis fails due to a *Shellzer*'s limitation, just in the cases labeled with E and F ($\sim 2.1\%$), while in the other two cases (labeled with C and D) the tool has been able to continue the analysis as far as even manual debugging has to stop.

5.2 Shellcode's database analysis

We now discuss what we found during the evaluation phase. The vast majority of the samples we analyzed, aim to retrieve and execute an additional payload.

In 3,124 cases, they try to retrieve more than one payload. Interestingly, in 1,445 cases the payload was still available, and *Shellzer* has been able to return its decrypted version. Furthermore, not always the resource downloaded is a malware in the form of a common Windows executable. Instead, in 898 cases, a malicious DLL is retrieved. We also found 2 samples that aim to execute HTA files (HTML applications) by launching `mshta.exe`, the HTA Windows interpreter. 3,429 samples allocate a new region of memory, they copy into it a portion of their own code, and then they jump there. Interestingly, some shellcode samples do this trick for up to six times in a row, and they download and execute an additional malware only in the last stage. Some samples contain two distinct shellcodes, where the first is in clear, while the second is heavily encrypted. This is probably done to mislead static analyzers that, after detecting and analyzing the first shellcode, will consider their job completed. In 1,327 cases, the shellcode calls the API #101 exported by `shdocvw.dll` (#101 is its ordinal number; it does not have an associated symbol). To the best of our knowledge, this should avoid the crash of the browser after the shellcode's execution. We also found 306 samples that make use of the UNICODE version of some Windows API functions: in these cases, the shellcode's complexity is increased, but it is more likely that the calls to those API functions are not monitored.

29 samples extracted from malicious PDF documents, perform the following additional operations (besides retrieving and running the malware): they first determine the path of the PDF Reader executable (through the `GetCommandLineA` API); they access again the malicious PDF document in order to extract a benign PDF document; they launch the PDF reader executable and, as an argument, they pass the path to the benign PDF document. In this way, the shellcode's execution, instead of ending with a crash, will result in a valid PDF document that is correctly opened, and this is often sufficient to fool an unsuspecting user.

We also found two samples that try to inject another shellcode in a different process, by properly using the `CreateRemoteThread` API function; 14 samples that try to perform a remote connection; and only 2 samples that interact with the Windows Registry. This shows how this kind of shellcode is infrequent when dealing with web-based malware and malicious PDF documents.

6 Related Work

Much work has been done that addresses the problem of shellcode's detection in a generic stream of bytes ([3, 14–16, 13, 11]), but relatively few attempts have been proposed that focus on shellcode's analysis.

One of the most popular way to analyze shellcode is to perform manual analysis by using debuggers (such as OllyDbg [20], Immunity Debugger [1], and WinDbg [2]) and static code analyzers, such as IDA Pro [12]. Unfortunately, this process is too slow to deal with a large number of samples, and it requires significant domain expertise. Furthermore, static analyzers cannot produce any meaningful results if strong encryption is used. This makes clear why automatic programs that dynamically analyze shellcode are needed.

A common way to dynamically analyze a binary consists in emulating its execution, by using tools like QEMU [8] or a library such as libemu [3]. Moreover, a tool named Spector [9] has been recently proposed: it focuses on shellcode's analysis and it uses an approach based on symbolic execution. These works carried big advantages with respect to manual analysis, but they suffer from some limitations. One of the issues is related with the big overhead introduced by emulation. This represents a problem since the execution of millions of instructions has to be emulated, due to the frequent presence of loops whose bodies are iterated thousands of times. Spector addresses this problem by using signatures that associate a known sequence of instructions to an equivalent high-level procedure, in a way that just such simple procedure is emulated, instead of emulating the execution of thousands of instructions. But if a simple unseen loop is introduced, this approach cannot give any performance speed-up. On the other hand, *Shellzer* does not have to use signatures to perform well. Indeed, thanks to the loop handler algorithm, our tool automatically switches from single-step instrumentation (that is analog to emulation) to normal execution, in a way that during the loops' iterations no overhead is introduced. Furthermore, Spector is limited to the execution of deterministic code, while *Shellzer* does not suffer from this problem since the shellcode is really executed. Another issue that affects some emulation-based approaches is that no real-data is downloaded from the network. This represents a problem, especially when analyzing samples extracted from web-based malware, since the shellcode must be able to access the retrieved binaries in order to be properly executed and analyzed. On the other hand, this kind of limitation provides some advantages. For example, while Spector can analyze shellcode samples that aim to open a remote command shell for an attacker, *Shellzer* is currently not able to do that. A simple solution would be to emulate the behavior of those specific API functions.

Another advantage of *Shellzer* with respect to other tools, is that we made possible to properly analyze shellcode samples that require to be executed in a specific environment, like the ones extracted from malicious PDF documents. This is why generic malware analyzers (like Anubis [4] and Threat Expert [6]) cannot provide accurate results in these cases.

7 Conclusion and Future Work

In this paper, we have presented *Shellzer*, a tool for the dynamic analysis of shellcode extracted from web-based malware and malicious PDF documents. Thanks to a series of optimizations, the single-step instrumentation turned out to be a successful approach, as confirmed by the high success rate achieved during the analysis of more than 24,000 real-world samples. As output, the tool returns a detailed report, which includes a complete trace of the API calls, and the payloads retrieved during the analysis (in their decrypted form). *Shellzer* also satisfies all the constraints required to properly analyze shellcode extracted from malicious PDF documents, and handle all the evasion techniques we found in the wild. In the near future, this tool will be integrated with Wepawet, and it will hence receive public attention. For this reason, our future work will focus on how shellcode samples could detect *Shellzer* and evade our analysis. Moreover, we are planning to introduce the possibility to analyze shellcode extracted from different sources, other than web-based malware and malicious PDF documents, and hence to extend the support for different execution contexts.

Acknowledgments. This work was partially supported by ONR grant N000140911042, ARO grant W911NF0910553, NSF grants CNS-0845559 and CNS-0905537, and Secure Business Austria.

References

1. Immunity debugger. `http://www.immunityinc.com/products-immdbg.shtml`
2. Ms debugging tools. `http://www.microsoft.com/whdc/devtools/debugging/`
3. Libemu. `http://libemu.carnivore.it` (2007)
4. Anubis. `http://anubis.iseclab.org` (2008)
5. Wepawet. `http://wepawet.cs.ucsb.edu/` (2008)
6. Threat expert. `http://www.threatexpert.com` (2009)
7. Amiri, P.: Pydbg. `http://pedram.redhive.com/PyDbg/` (2005)
8. Bellard, F.: Qemu. `http://www.qemu.org` (2005)
9. Borders, K., Prakash, A., Zielinski, M.: Spector: Automatically analyzing shell code. In: Proceeding of the Annual Computer Security Applications Conference (ACSAC) (2007)
10. Cova, M., Kruegel, C., Vigna, G.: Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In: Proceedings of the World Wide Web Conference (WWW) (2010)
11. Gu, B., Bai, X., Yang, Z., Champion, A.C., Xuan, D.: Malicious shellcode detection with virtual memory snapshots. In: Proceedings of the IEEE International Conference on Computer Communications (INFOCOM) (2010)
12. Hex-Rays: Ida pro disassembler and debugger. `http://www.hex-rays.com/idapro/`
13. Payer, U., Teufl, P., Lamberger, M.: Hybrid engine for polymorphic shellcode detection. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) (2005)
14. Polychronakis, M., Anagnostakis, K., Markatos, E.: Network-level polymorphic shellcode detection using emulation. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) (2006)
15. Polychronakis, M., Anagnostakis, K., Markatos, E.: Emulation-based detection of non-self-contained polymorphic shellcode. In: Proceedings of Recent Advances in Intrusion Detection (RAID) (2007)
16. Polychronakis, M., Anagnostakis, K., Markatos, E.: Comprehensive shellcode detection using runtime heuristics. In: Proceeding of the Annual Computer Security Applications Conference (ACSAC) (2010)
17. Shields, T.: Anti-debugging - a developers view
18. Singh, A.: Identifying Malicious Code Through Reverse Engineering. Springer (2009)
19. Skape: Understanding windows shellcode. `http://www.hick.org/code/skape/papers/win32-shellcode.pdf`
20. Yuschuk, O.: Ollydbg. `http://www.ollydbg.de/` (2005)