Assisting Malware Analysis with Symbolic Execution: A Case Study

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia(✉), and Camil Demetrescu

Software Analysis and Optimization Laboratory,
Department of Computer, Control, and Management Engineering,
Cyber Intelligence and Information Security Research Center,
Sapienza University of Rome, Rome, Italy
{baldoni,coppa,delia,demetres}@dis.uniroma1.it

Abstract. Security analysts spend days or even weeks in trying to understand the inner workings of malicious software, using a plethora of manually orchestrated tools. Devising automated tools and techniques to assist and speed up the analysis process remains a major endeavor in computer security. While manual intervention will likely remain a key ingredient in the short and mid term, the recent advances in static and dynamic analysis techniques have the potential to significantly impact the malware analysis practice. In this paper we show how an analyst can use symbolic execution techniques to unveil critical behavior of a remote access trojan (RAT). Using a tool we implemented in the Angr framework, we analyze a sample drawn from a well-known RAT family that leverages thread injection vulnerabilities in the Microsoft Win32 API. Our case study shows how to automatically derive the list of commands supported by the RAT and the sequence of system calls that are activated for each of them, systematically exploring the stealthy communication protocol with the server and yielding clues to potential threats that may pass unnoticed by a manual inspection.

Keywords: Malware · RAT · APT · Symbolic execution · Angr

1 Introduction

The unprecedented spread of network-connected devices and the increasing complexity of operating systems is exposing modern ICT infrastructures to malicious intrusions by different threat actors, which can steal sensitive information, gain unauthorized access, and disrupt computer systems. Attacks are often perpetrated in the context of targeted or broad-spectrum campaigns with different scopes, including hacktivism, cyber warfare, cyber crime, and espionage. One of the most common form of intrusion is based on malicious software, or malware, which can exploit vulnerabilities in applications and operating systems to infect, take over, or disrupt a host computer without the owner's knowledge and consent. Sustained by the explosion of messaging applications and social

S. Dolev and S. Lodha (Eds.): CSCML 2017, LNCS 10332, pp. 171–188, 2017.
DOI: 10.1007/978-3-319-60080-2_12

networks, malware can nowadays affect virtually any device connected to the Internet including unconventional targets such as network printers, cooling systems, and Web-based vehicle immobilization systems. Malware infections can cause potentially significant harm by exfiltrating sensitive data, tampering with databases and services, and even compromising critical infrastructures.

According to [17], malware is responsible for the most frequent and costly attacks on public and private organizations. ICT infrastructures are not the only targets: Kindsight Security reports that at least 14% of private home networks were infected with malware in April–June 2012 [16]. One of the main vectors of malware spreading remain emails and infected websites, where unsuspecting users are daily hijacked by inadvertently opening seemingly benign attachments or lured into browsing deceitful links or click-baits that stealthily download and install malicious software. Malware scammers often resort to social engineering techniques to trick their victims and infect them with a variety of clever approaches including backdoors, trojans, botnets, rootkits, adware, etc.

The job of a professional malware analyst is to provide quick feedback on security incidents that involve malicious software, identifying the attack vectors and the proper actions to secure and repair the infected systems. In many cases involving critical compromised services, time is of the essence. Analysts seek clues to the parts of the system that were disrupted and attempt to reconstruct and document the chain of events that led to the attack. Often, intrusions are carried out by variants of previously encountered malware. In other cases, malware is based on zero-day vulnerabilities or novel attack strategies that may require days or even weeks to be identified and documented. Analysts usually combine and relate the reports generated by a wide range of dedicated static and dynamic analysis tools in a complex manual process and are often required to sift through thousands or even millions of lines of assembly code.

A skilled professional is able to glance over irrelevant details and follow the high-level execution flow, identifying any stealthy API calls that can compromise the system. However, some tasks may keep even the most experienced analysts busy for days or even weeks. For instance, malware such as backdoors or trojans provide a variety of hidden functionalities that are activated based on unknown communication protocols with remote servers maintained by the threat actors. Identifying the supported commands and reconstructing how the protocols work may require exploring a wide range of possible execution states and isolating the input data packets that make the execution reach them. While this can be amazingly difficult without automated software analysis techniques, the state of the art of modern binary reverse engineering tools still requires a great deal of manual investigation by malware analysts.

Contributions. In this paper, we argue that the significant advances in software analysis over the last decade can provide invaluable support to malware analysis. In particular, we describe how symbolic execution [1], a powerful analysis technique pioneered in software testing, can be applied to malware analysis by devising a prototype tool based on the Angr symbolic executor [26]. The tool automatically explores the possible execution paths of bounded length starting

from a given entry point. The analysis is static and the code is not concretely executed. As output, the tool produces a report that lists for each explored execution path the sequence of encountered API calls and their arguments, along with properties of the malware's input for which the path is traversed, e.g., the actual data values read from a socket that would trigger the path's execution.

We evaluate our tool on a sample taken from a well-known family of remote access trojans [31], showing how to automatically reconstruct its communication protocol and the supported commands starting from initial hints by the malware analysts on the portions of code to analyze. We estimate the reports our tools generates to be worth hours of detailed investigation by a professional analyst.

Paper organization. This paper is organized as follows. In Sect. 2 we address the background and the features of our case study. Section 3 discusses our symbolic analysis tool and how we used it to analyze the sample. Section 4 puts our results into the perspective of related work, while Sect. 5 concludes the paper with final thoughts and ideas for further investigations.

2 Features of the RAT

In this section, we discuss the background and the features of the malware instance we use as our case study. The RAT executable can be downloaded from VirusTotal and its MD5 signature is `7296d00d1ecfd150b7811bdb010f3e58`. It is drawn from a family of backdoor malware specifically created to execute remote commands in Microsoft Windows platforms.

Variants of this RAT date as far back as 2004 and have successfully compromised thousands of computers across more than 60 different countries. This constantly morphing malware is known under different aliases such as Enfal and GoldSun, and its instances typically contain unique identifiers to keep track of which computers have been compromised by each campaign [31].

The RAT gathers information on the infected computer, and communicates with a command-and-control (C&C) server. Once activated, the malware allows a remote attacker to take over the infected machine by exchanging files with the server and executing shell commands. It then copies itself in a number of executable files of the Windows system folder and modifies the registry so that it is automatically launched at every startup.

The malware uses thread injection to activate its payload in Windows Explorer. The payload connects to http://mse.vmnat.com, sending a `sprintf`-formatted string with information on the system retrieved using the Netbios API. At the core of the malware is a loop that periodically polls the server for encrypted remote commands and decrypts them by applying a character-wise XOR with the 0x45 constant. The malware version we analyzed supports 17 commands, which provide full control over the infected host by allowing remote attackers to list, create, and delete directories, move, delete, send, and receive files, execute arbitrary commands, terminate processes, and other more specific tasks. Communication with the server is carried out over HTTP 1.1 on port 80, user "reader", and password "1qazxsw2".

A high-level picture of the control flow graph of the command processing thread, automatically reconstructed by the IDA disassembler, is shown in Fig. 1. The thread code starts at address 0x402BB0 and spans over 4 KiB of machine code, not including the code of the called subroutines.

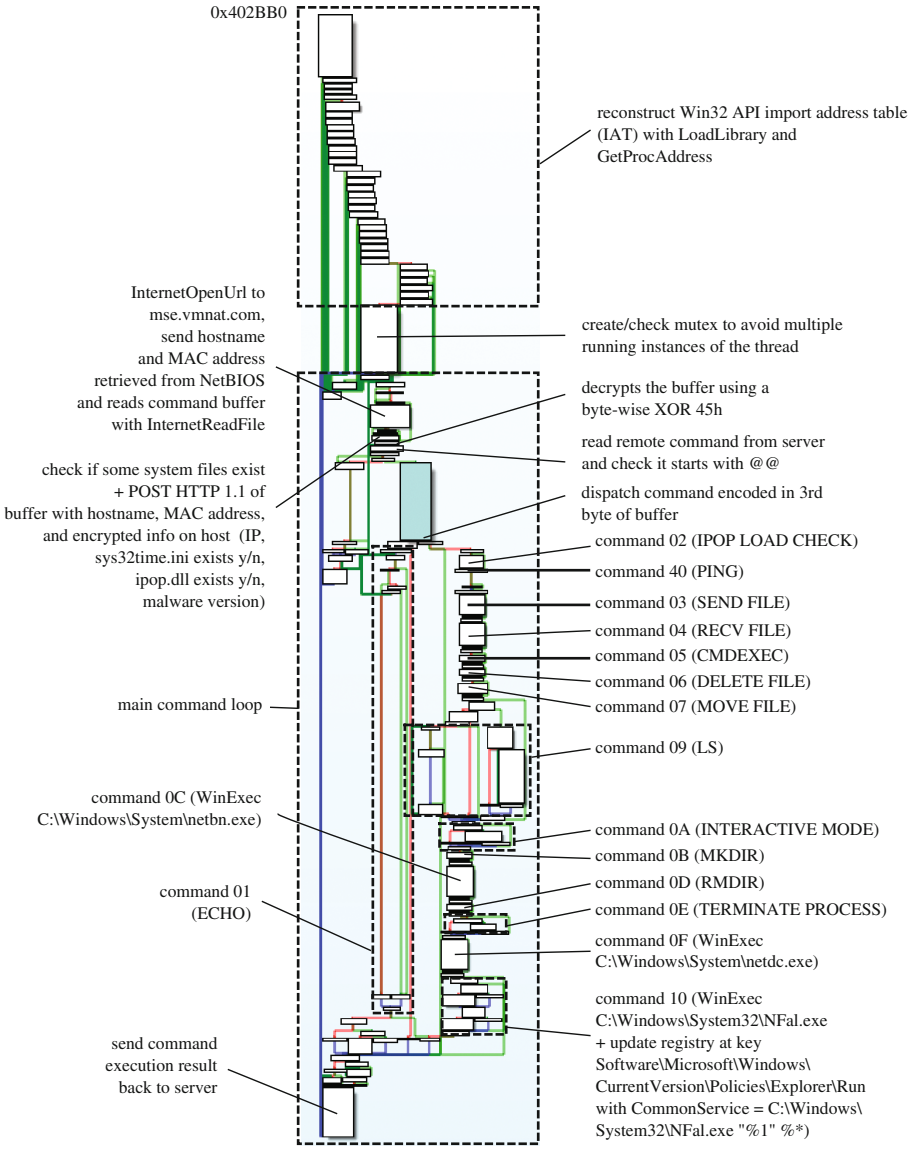

Fig. 1. Control flow graph of the RAT's command processing thread.

3 Analyzing the RAT with Symbolic Execution

In this section, we report our experience in the analysis of our RAT sample using Angr, a popular symbolic executor. After a brief description of symbolic execution techniques, we discuss the practical challenges in using these tools for the malware realm, and present a number of domain-specific optimizations we adopted. We describe the setup we used to dissect the core of the sample, and discuss our findings and their value from a malware analyst's perspective.

3.1 Introducing Symbolic Execution

Symbolic execution is a popular program analysis technique for testing a property in program against multiple execution paths at a time. Rather than executing a program on a specific input – thus exploring a single control flow path at a time – symbolic execution can concurrently handle multiple paths that would be exercised under different inputs.

In a nutshell, a program is allowed to take on *symbolic* rather than concrete input values, while an execution engine collects across each explored path a set of constraints that are combined into a formula describing the path. When an assignment instruction is evaluated, the formula is simply updated to reflect it. When a branching instruction is encountered and its outcome depends on one or more symbolic values, the execution is forked by creating two states described by two distinct formulas, derived from the current formula by adding to it the branch condition or its negation, respectively. A constraint solver - typically one suited for satisfiability modulo theories (SMT) - is used to evaluate expressions involving symbolic values, as well as for generating concrete inputs that can be used to run concretely the program along the desired path.

We have implemented our ideas in Angr [26], an open source framework for symbolic execution developed at UC Santa Barbara. Angr ships as a collection of Python libraries for binary analysis and dissection. It has been employed in a number of research works [25,29], as well as by the Shellphish team from UCSB that recently participated in the DARPA Cyber Grand Challenge, a two-year competition seeking to create automatic systems for vulnerability detection, exploitation, and patching in near real-time [26]. The simplicity of its Python APIs and the support provided by the community make it an ideal playground for prototyping research ideas in a powerful framework.

3.2 Addressing Challenges from the Malware Domain

Symbolic execution techniques have largely been employed in the software testing domain, usually with the goal of automatically generating test inputs that yield a high degree of code coverage. Albeit a few works have explored security applications of these techniques (Sect. 4), the majority of currently available symbolic executors are not well equipped for analyses in the malware realm.

Firstly, most research tools target Linux as a platform, while Windows is by far the most malware-ridden platform. The few tools that support Windows

APIs such as BitBlaze [28] commonly resort to a concrete execution for API calls, asking the constraint solver to provide a valid model for a path formula when a symbolic argument is provided for a call. This *concretization* process typically causes a loss of accuracy in the analysis; also, it does not solve scalability problems that might arise when analyzing real-world malware [30]. Secondly, techniques and heuristics designed for software testing purposes might not fit malware well. While in software testing it is profitable to explore paths capturing behaviors unexpected for a standard usage session, such as system or library call failures, in the malware domain an analyst is rather interested in behaviors commonly exposed by a sample, provided that the right triggering inputs (e.g., valid commands from a C&C server) are received and/or the environmental conditions (e.g., a specific version of Windows) are met.

Extending Angr**.** To make the dissection of the RAT possible, we had to devise a number of extensions to the original Angr framework, tackling both of the above-mentioned problems. In order to support Windows APIs, we had to implement 57 models of commonly used functions, such as `GetProcAddress`, `LoadLibrary`, `HttpOpenRequest`, and `CreateFile`. A *model* is a summary for a function that simulates its effects by propagating symbolic data in the same way that the original function would have [3], requiring a significantly shorter amount of time than in a symbolic exploration. Models are commonly employed in symbolic executors when dealing with the environment (e.g., to simulate filesystem and network I/O) and also to speed up the analysis of classes of functions, such as those for string manipulation. The lack of models for Win32 APIs in Angr was thus the first obstacle we had to overcome, along with the missing support for the `stdcall` calling convention employed by these APIs.

Writing a model can be a time-consuming and error-prone task [2]. We thus developed a tool that fetches API prototypes from Windows header files and retrieves from the MSDN documentation not only a description of the function's behavior, but also of the directional attribute (i.e., *in*, *out*, or both) of each argument. The output of the tool is thus an Angr model stub annotated with the input-output relationships, so that a developer can more easily encode the logic of the function in terms of symbolic data manipulation.

Domain-Specific Optimizations. We then had to assess the practicality of our methodology with respect to the so-called *path explosion* problem, which haunts every symbolic execution implementation and can prevent it from scaling to real-world software. Indeed, as a symbolic executor may fork states at every input-dependent branch, the total number of paths to explore might be exponential. This impacts both space and time, and a common approach is to employ search strategies that can limit the exploration to a subset of paths that look appealing for a given goal (e.g., identifying null-pointer dereferences).

The adoption of domain-specific optimizations and search heuristics can mitigate the path explosion problem in the analysis of malicious binaries, making their symbolic analysis feasible. A recent work [30] explores this approach for binaries packed with sophisticated code packers that reveal pages only when about to execute code in them. We thus devised a number of heuristics

aiming at revealing information useful to an analyst for the dissection, discarding instead from the exploration paths that are unlikely to.

For a number of models, we restricted the number of possible outcomes by discarding error paths, or put a limit on the length of the symbolic buffer returned by a method. For instance, in our case study we found fairly reasonable to assume that having Win32 methods such as `HttpSendRequest` or `InternetReadFile` to succeed should fairly reflect the run-time behavior expected for a malware. Shall an adversary put a number of faulty function invocations in the code as a decoy, the analyst can still revert the model to an exhaustive configuration (either at specific call sites or for any invocation) and restart the symbolic execution. Exploring error-handling paths might become necessary for malware, especially nation-state APTs, that conceals interesting behavior for the security research to explore, e.g., attempting to spread the infection to other computers until Internet access is found on one of them. Selectively enabling error-handling paths provides the analyst with the flexibility needed to explore such scenarios as well.

Limiting the length of symbolic buffers is a common practice in symbolic executors, as exploring all possible lengths would quickly lead to the path explosion phenomenon [1]. We devised an optimization that targets symbolic buffers originating in Win32 models and processed by tight loops. A *tight loop* is a code sequence ending in a conditional backward jump within a short reach; for our setting we empirically chose a 45-byte threshold. When the loop trip count depends on the content of a symbolic buffer originating in a Win32 model, we do as follows: if the loop has not already exited within k iterations, we force it to. The rationale behind this heuristic is that while such buffers are typically large, the amount of data they usually contain is much smaller. For instance, in the analysis of our RAT sample we found out that most buffers have length `0x1000` but are filled only for the first few bytes. Tight loops for string processing are quite frequent in the sample, especially in the form of `REP` instructions. We also provided an optimization for Angr that speeds up the processing of `REP` when the trip count can be determined statically: rather than symbolically iterating over the loop, we compute its effects and update the state at once.

Finally, we encoded a simple iterative deepening search (IDS) strategy to discriminate which alternative should be explored first when a branching instruction is encountered (Sect. 3.1). As our goal is to reconstruct which strings exercise the different commands supported by a RAT and possibly discover any dependencies between them, exploring sequences of commands of increasing length might provide insights to an analyst in a timely manner. We also favored IDS over breadth-first search (BFS) for its lower memory consumption. In fact, Angr currently lacks a mature checkpointing mechanism to automatically suspend the exploration for a set of states and release their resources when too many paths are being concurrently explored. While this might not be an issue when executing the framework on a cluster, an analyst might also want to perform a preliminary analysis on commodity hardware such as the laptop we used in our experiments.

3.3 Dissecting the RAT with Angr

In this section, we provide an overview of how the dissection of the RAT sample can be carried out in our tool for Angr.

When a malware analyst first disassembles the executable, they can determine from the inspection of the `WinMain` method that the RAT - after running a number of checks and collecting preliminary information about the attacked machine (such as the Windows version and the locale in use) - resorts to the Win32 `CreateRemoteThread` function to inject and run three threads in the virtual address space of `explorer.exe`. A quick look at their code reveals that the first thread carries out the command-and-control logic and executes the remote commands (Fig. 1), while the remaining two do not reveal any interesting behavior and can be discarded from the analysis for the moment.

Execution Context. Ideally, an analyst would like to set the symbolic entry point (SEP) to the entry point of the thread and start the symbolic execution from there. Angr treats as symbolic any data fetched from uninitialized memory locations during the exploration. We thus define the *execution context* as the set of memory location holding a value initialized when reaching SEP from the program's entry point, and that any path starting at SEP might then need later[1].

Providing the engine with information on the context can be profitable depending on the application being analyzed. For instance, one might employ symbolic execution to find out how the context should look like in order for the execution to reach a specific instruction (e.g., to discover which input string defuses a logic bomb). For other applications, however, a fully symbolic context might quickly lead to an explosion of the paths, as too little information is available when dealing with assignments and branching instructions.

In our case study, the execution context for the command processing thread consists of a large array provided as argument to the thread. This array contains gathered information describing the attacked machine, the addresses of the `LoadLibrary` and `GetProcAddress` functions in the address space of the process to inject, and a number of strings describing the names of the Win32 APIs that will be used in the thread. In fact, when executing in the injected process the RAT will solve the address of each Win32 function it needs to invoke in a block of code, constructing on the stack a minimal import table for the block on demand.

It is unrealistic to think that in general a constraint solver can guess which API a malware writer requires at specific points in the program. The analyst in this scenario is thus expected to provide the symbolic executor with portions of the context, i.e., the API-related strings added to the argument array in the early stage of a concrete execution. This should not be surprising for an analyst, as they often have to fix the program state when manipulating the stack pointer in a debugger to advance the execution and skip some code portions. We have explored two ways to perform this task. The first is to take a memory dump of a concrete execution of the program right before it reaches the starting point SEP

[1] A context can formally be defined in terms of live variables, i.e., the set of locations that execution paths starting at SEP might read from before writing to.

for the symbolic execution. A script then processes it and fills the context for the execution with concrete values taken from the dump. Optionally, portions of the context can be overridden and marked as symbolic: for instance, turning into symbolic a buffer containing the IP address or the Windows version for the machine can reveal more execution paths if a malware discriminates its behavior according to the value it holds.

The problem with a dump-based approach is that in some cases it might not be simple for an analyst to have a concrete execution reach the desired SEP. A more general alternative is to employ symbolic execution itself to fill a portion of the context, by moving the SEP backward. In our RAT case study we symbolically executed the instructions filling the argument array, obtaining a context analogous to what we extracted from a memory dump.

We believe that in general a combination of the two approaches might help an analyst reconstruct the context for even more sophisticated samples, if it is required by the executor to produce meaningful results. Additionally, reverse engineering practitioners that use symbolic execution often perform simple adjustments on an initial fully symbolic context in order to explore complex portions of code. We believe this approach can effectively be applied to samples from the malware domain as well. In fact, such adjustments are common in the Angr practice as part of a trial-and-error process, and do not require intimate knowledge of the inner workings of the framework.

Starting the Exploration. Our RAT dissection tool ships as an Angr script that we devise in two variants: one takes on a concrete instantiation of the argument array for the thread, while the other constructs it symbolically. Nevertheless, once the entry point of the injected thread is reached, their behavior is identical: from here on we will thus use the term *script* to refer to both of them. From the thread's entry point the symbolic execution follows a single execution path until a `while` cycle is reached. As we will discuss in the next section, this cycle is responsible for command processing. The analyst would then observe in the run-time log of the script that a symbolic buffer is created and manipulated in Internet-related Win32 API calls (e.g., `InternetReadFile`), and different execution paths are then explored depending on its content.

The analyst can ask the tool to display how the branch condition looks like. The language used for expressing the condition is the one used by the Z3 SMT solver employed by Angr. If the condition is hardly readable by the analyst, the script can also query the constraint solver and retrieve one or more concrete instances that satisfy the condition. Our sample initially checks whether the first two bytes in a symbolic buffer are equal to a constant that when subsequently XOR-ed with the encryption key yields the sequence "`@@`".

As the exploration proceeds, paths in which the condition is not satisfied will quickly return to the beginning of the `while` cycle. This might suggest that the constraints on the content of the symbolic buffer do not match the syntax of the command processing core, which would then wait for a new message.

The analyst can also employ similar speculations or findings to speed up the symbolic execution process. In particular, ANGR allows users to mark certain addresses as to avoid, i.e., the engine will not follow paths that bring the instruction pointer to any of them. For the sample we dissected this was not a compelling issue: the number of paths that our iterative deepening search (IDS) strategy would explore would still be small. Nonetheless, this optimization can become very valuable in a scenario where the number of paths is much larger due to a complex sequence of format controls. A direct consequence of the optimization is that paths spanning sequences with at least one invalid command in it are not reported to the analyst. We believe that such paths would not normally reveal valuable insights into the command processing logic and protocol, and thus can safely be discarded.

While attempting to dissect the command processing loop, the IDS strategy used for path exploration (Sect. 3.2) has proved to be very convenient. In fact, an IDS allows us to explore possible sequences of commands of increasing length k, which also corresponds to the number of times the first instruction in the `while` cycle is hit again. The script will produce a report every time a sequence of k commands is fully explored, and once all possible sequences have been analyzed the symbolic executor proceeds by exploring a sequence of $k + 1$ commands, producing in turn incremental reports for the updated length. As the number of iterations of a command processing loop is typically unbounded, the analyst has to explicitly halt the exploration once they have gained sufficient insights for the objective of the analysis. For our sample, all the accepted commands were already revealed for $k = 3$.

3.4 The RAT Dissected

The reports generated from our tool capture a number of relevant and useful facts to an analyst regarding each execution path. Each report is a polished version of the execution trace revealing the sequence of Win32 API invocations performed inside each x86 subroutine in the sample. A report also captures control flow transfers across subroutines, showing their call sites and return addresses along with the API calls they perform in between.

```
...
[0x4030a0] InternetOpenA(<BV32 0x0>, <BV32 0x0>, <BV32 0x0>, <BV32 0x0>, <BV32 0x0>)
    => <BV32 hInternet_39_32>
[0x4030bc] InternetOpenUrlA(<BV32 hInternet_39_32>, <BV32 0xabcd161c>,
    <BV32 0x0>, <BV32 0x0>, <BV32 0x84000100>, <BV32 0x0>,
    'http://mse.vmnat.com/httpdocs/mm/$machine_host_name:$mac_address/Cmwhite')
    => <BV32 hInternet_url_40_32>
[0x4030d5] InternetReadFile(<BV32 hInternet_url_40_32>, <BV32 0x7ffd4c00>,
    <BV32 0x1000>, <BV32 0x7ffd4a60>) => <BV32 0x1>
    S0: <BV32768 InternetReadFile_buffer_41_32768> @ 0x7ffd4c00
    S0: <BV32 InternetReadFile_buffer_written_42_32> @ 0x7ffd4a60
[0x4030dc] InternetCloseHandle(<BV32 hInternet_url_40_32>) => <BV32 0x1>
[0x4030df] InternetCloseHandle(<BV32 hInternet_39_32>) => <BV32 0x1>
...
```

Fig. 2. Fragment of detailed report automatically generated for one execution path in the RAT's command processing thread.

Figure 2 shows an excerpt from a report. For each API the call site, the list of arguments and the return value are shown. Constant string arguments are printed explicitly, while for other data types we resort to the `BVxx` notation, which in the Angr domain describes a *bitvector* (i.e., a sequence of consecutive bits in memory) of `xx` bits. `BV32` objects occur frequently on a 32-bit architecture as they can hold pointers or primitive data types. When a bitvector holds a concrete value, the value is explicitly listed in the report. For symbolic values a string indicating a data type (e.g., `hInternet_url`) or the name of the API that created the buffer (e.g., `InternetReadFile_buffer`) is displayed, followed by a suffix containing a numeric identifier for the buffer and the buffer size in bits. Observe that while the contents of a bitvector can be symbolic, it will normally be allocated at a concrete address: when such an address is passed as argument to an API, the report will contain a row starting with `SO:` that describes the symbolic buffer the address points to.

We remark that the sequence of Win32 API calls performed by a malware typically provides valuable information in the malware analysis practice, as the analyst can find out which are the effects of a possible execution path in a black-box fashion. We will further discuss this aspect in Sect. 4.

Further insights can be revealed once the constraint solver produces for the report instances of symbolic buffers matching the path formula (Sect. 3.1), i.e., inputs can steer the execution across the analyzed path. For instance, this allowed us to find out which strings were required to exercise the 17 commands accepted by our sample (Fig. 1).

Dependencies between commands can instead surface from the analysis of sequences of increasing length k. We found that each valid sequence of commands should always start with two invocations of the 01 command, used to implement a handshaking protocol with the command and control server.

From a concrete instantiation of the symbolic buffers we then found out the RAT checks for the presence of a magic sequence in the server's response during the handshaking phase. In particular, the response has to contain "8040$(" starting at byte 9 in order for the malware to update its state correctly and eventually unlock the other commands. Constraint solving techniques thus proved to be valuable in the context of message format reconstruction.

Sequences of three commands reveal sufficient information for an analyst to discover all the commands accepted by the RAT. Due to the particular structure of the handshaking protocol, our tool explored (and thus reported) as many paths as supported commands. Figure 3 provides a compact representation of the logic of the command processing thread that we could infer from the reports. The sample starts by creating a mutex to ensure that only one instance of the RAT is running in the system. The internal state of the malware, represented by the two variables `c1` and `c2`, is then initialized.

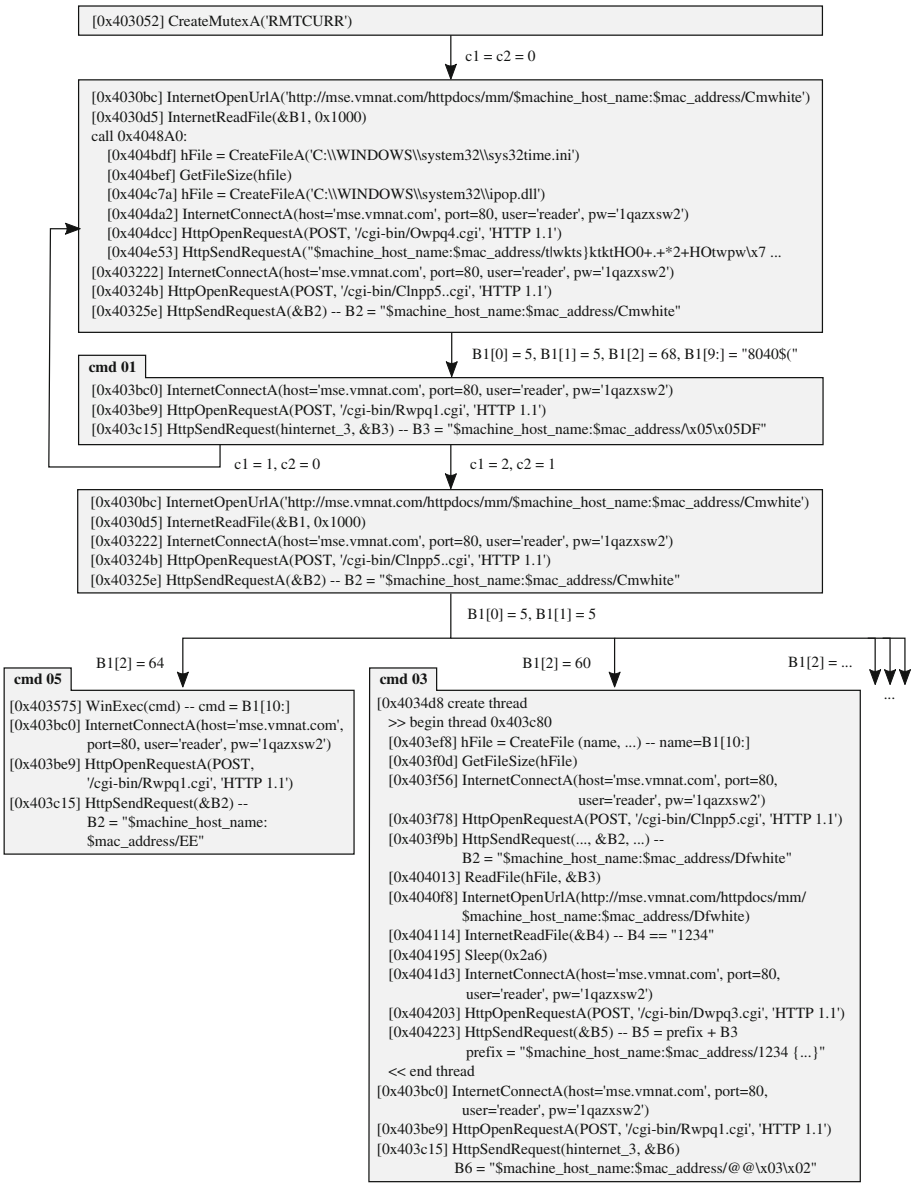

Fig. 3. Compact report for the RAT's command processing thread.

The subroutine starting at address 0x4048A0 is invoked to collect further information on the machine (specifically, the presence of two files), and a connection is established with the server in order to transmit the identity of the infected machine. This step is performed twice, resulting in different increments to `c1` and `c2`. Edges between code blocks have been annotated with the

conditions on the symbolic bytes in the response from the server that should be met in order to make the transitions possible.

Once `c1` = 2 and `c2` = 1, the whole set of commands for the sample is unlocked. Figure 3 reports a high-level description of two commands: namely, command 05 executes the *cmd* application on the machine, while command 03 spawns a thread to transmit a file to the server. Both command handlers end with a sequence that notifies the server of the reception of the command. We extracted from the reports the sequence of Win32 API calls performed in each command, thus identifying all the actions available to the attackers.

4 Related Work

Malware Detection. Anti-malware vendors receive every day thousands of samples, many of which are unknown. A large body of works have explored automatic techniques to determine whether a sample is malicious and, if so, whether it is a variation of a previously analyzed threat or it requires a closer inspection from a malware analyst. Solutions based on static techniques analyze the code without actually executing it, with the advantage of covering it in its entirety. For instance, [8] relies on model checking to defy simple obfuscations employed by malicious code writers to subvert detection tools such as anti-virus products. [9] extends this technique by supporting a wider range of morphing techniques common in polymorphic and metamorphic malware.

The major weakness of static solutions is that they can be defeated by resorting to self-modifying code, as in packer programs, or to techniques designed to foil static disassembly. Dynamic solutions are thus a necessary complement to static detection techniques [18]. Dynamic techniques typically execute a sample in a contained environment and verify the action it performs, providing analysts with a report. For instance, GFI Sandbox [33] and Norman Sandbox [27] are popular tools among security professionals. Dynamic analyses can monitor a number of aspects including function calls and their arguments, information flow, and instruction traces. We refer the interested reader to [12] for a recent survey of this literature. The main drawbacks of dynamic solutions are that only a single program execution is observed, and that a malware might hide its behavior once it detects it is running in a contained environment.

Automatic Code Analysis. A few works have attempted to automatically explore multiple execution paths for a malware sample. In [18] Moser *et al.* present a system that can identify malicious actions carried out only when certain conditions along the execution paths are met. Their results show that on a set of 308 real-world malicious samples many of them show different behavior depending on the inputs from the environment. Brumley *et al.* [3,4] have designed similar systems aiming at identifying trigger-based behavior in malware. In particular, [3] discovers all commands in a few simple DDoS zombies and botnet programs.

These approaches employ mixed concrete and symbolic execution to explore multiple paths, starting the execution from the very beginning of the program. In this paper, we leverage symbolic execution to dissect a portion of a sample that is of interest for an analyst, provided they have sufficient knowledge to set up a minimal execution context for the exploration to start. Automatic systems suffer from known limitations that hinder the analysis of complex malware, such as the inherent cost of constraint solving and difficulties in handling self-modifying code and obfuscated control flow [20]. They are thus not generally used for real-scale malware analysis [30]. We believe manual symbolic execution as devised in this work can help get around these issues, as an analyst can provide the engine with insights to refine and guide the exploration (on a possibly limited scope compared to a whole-code automatic analysis) as part of a trial-and-error process.

In [30] Ugarte-Pedrero *et al.* show that by leveraging a set of domain-specific optimizations and heuristics multi-path exploration can be used to defeat complex packers. They also present an interesting case study on Armadillo, which is very popular among malware writers. Ad-hoc techniques and heuristics, including even simple ones as those we describe in this paper, can indeed be very effective in the malware domain.

Of a different flavor is the framework presented in [20]. X-Force is a binary analysis engine that can force a binary to execute requiring no input or proper environment. By trading precision for practicality, branch outcomes are forced in order to explore multiple paths inconsistently, while an exception recovery mechanism allocates memory and updates pointers accordingly. In one of the case studies the authors discuss the discovery of hidden malicious behaviors involving library calls.

Symbolic Execution. Symbolic execution techniques have been pioneered in the mid 1970s to test whether certain properties can be violated by a piece of software [15]. Symbolic techniques have been largely employed in software testing, with the goal of finding inputs that exercise certain execution paths or program points (e.g., [6,13]). A number of security applications have been discussed as well, e.g., in [25,26,28,29]. The reader can refer to previous literature (e.g., [1,23]) for a better understanding of the challenges that affect the efficiency of symbolic execution and when it might become impractical.

To the best of our knowledge, symbolic execution tools are not commonly employed yet by malware analysts. However, the 2013 DARPA announcement regarding the Cyber Grand Challenge competition has raised a lot of interest among security professionals. For instance, the 2016 Hex-Rays plugin contest for IDA Pro was won by Ponce [14], which provides support for taint analysis [23] and symbolic execution: Ponce allows the user to control conditions involving symbolic registers or memory locations, in order to steer the execution as desired. Symbolic execution is employed also in several open-source projects such as Triton [22] for binary analysis, reverse engineering, and software verification.

Obfuscation. Obfuscation techniques can be used also with the specific goal of thwarting symbolic execution. In particular, [24] discusses how to use cryptographic hash functions to make it hard to identify which values satisfy a branch condition in a malware, while [32] relies on unsolved mathematical conjectures to deceive an SMT solver. [34] addresses the limitations of symbolic execution in the face of three generic code obfuscations and describes possible mitigations.

Botnet Analysis. In this paper, we show how to derive the sequence of commands for a specific RAT. Many works have tackled the more general problem of automatic protocol reconstruction and message format reverse engineering (e.g., [5,7,10,11]). We refer the interested reader to [19] for a survey of protocol reverse engineering techniques, and to [21] for a taxonomy of botnet research.

5 Conclusions

In this paper we have shown a successful application of symbolic execution techniques to malware analysis. A prototype tool we designed based on Angr was able to automatically derive the list of commands supported by a well-known RAT and its communication protocol with the C&C server. To design our tool we had to overcome a number of complex issues. A primary requirement for symbolically executing a Windows binary is the availability of API models. Unfortunately, the current release of Angr does not provide any Win32 API model, forcing us to develop them when needed for executing our RAT sample. An interesting research direction is how we can extend our tool for generating API stubs in order to minimize the implementation effort required to transform these stubs into working API models.

The most common issue when performing symbolic execution of a complex binary is the well-known *path explosion* problem. Indeed, a large number of paths could be generated during a program's execution, making the analysis hardly scalable. To mitigate this problem, we have implemented several domain-specific optimizations as well as a variety of *common-sense* heuristics. Although these tweaks may harm the efficacy of an automatic analysis by discarding potentially interesting paths, they can be easily disabled or tuned by a malware analyst whenever few useful reports are generated by our tool.

The case study presented in this paper has shown how communication protocols used by RATs could be potentially nontrivial. In the examined RAT sample, our tool was able to highlight that a specific handshaking phase is required to activate the majority of commands. While an analyst may spend hours trying to understand this protocol, our tool could reveal it without any manual intervention. However, our prototype still lacks support for mining the reports. Ideally, our tool should continuously evaluate the generated reports and provide the analyst with a clear summary of the findings, possibly highlighting and clustering reports based on common features. Visualizing the flow of concrete and symbolic data across API calls would provide valuable information to analysts as well.

While our prototype has been tested only on a single RAT sample, we believe our approach is rather general and replicable on other well-known RAT families.

We plan to address this topic in future work. One obstacle to a large-scale validation is that each sample may need a different setup, i.e., a different symbolic entry point and execution context. It remains an interesting open question how to minimize the amount of manual intervention required for malware analysts.

Another challenging issue that is likely to emerge when approaching other RAT samples is the use of strongly encrypted commands. Indeed, if a RAT resorts to a robust crypto function to decrypt the command, the constraint solver may be unable to provide a concrete model, i.e., break the encryption schema. In this scenario, our tool may fail to fully reconstruct the communication protocol of the RAT, but may still provide useful hints for the analyst. Although this may seem a critical limitation of our tool, we remark that, when performing a manual dissection, the analyst will face the same issue. Common crypto attacks, such as dictionary-based and brute-force attacks, could be integrated in our tool to attempt to defeat the encryption when the solver fails.

Acknowledgments. We are grateful to the anonymous CSCML 2017 referees for their many useful comments. This work is partially supported by a grant of the Italian Presidency of Ministry Council and by CINI Cybersecurity National Laboratory within the project "FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks" (www.filierasicura.it) funded by CISCO Systems Inc. and Leonardo SpA.

References

1. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. CoRR, abs/1610.00502 (2016)
2. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, EuroSys 2006, pp. 73–85. ACM, New York (2006)
3. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: Bitscope: automatically dissecting malicious binaries. Technical report, CMU-CS-07-133 (2007)
4. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Lee, W., Wang, C., Dagon, D. (eds.) Botnet Detection, pp. 65–88. Springer, Boston (2008)
5. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 317–329. ACM, New York (2007)
6. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008, pp. 209–224. USENIX Association, Berkeley (2008)
7. Cho, C.Y., Shin, E.C.R., Song, D.: Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 426–439. ACM, New York (2010)

8. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th Conference on USENIX Security Symposium, SSYM 2003, vol. 12. USENIX Association, Berkeley (2003)
9. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, SP 2005, pp. 32–46. IEEE Computer Society, Washington, DC (2005)
10. Cui, W., Kannan, J., Wang, H.J.: Discoverer: automatic protocol reverse engineering from network traces. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, SS 2007, pp. 14:1–14:14. USENIX Association, Berkeley (2007)
11. Cui, W., Peinado, M., Chen, K., Wang, J.H. and Irun-Briz, L.: Automatic reverse engineering of input formats. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 391–402. ACM, New York (2008)
12. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2), 6:1–6:42 (2008)
13. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2008 (2008)
14. Illera, A.G., Oca, F.: Introducing ponce: one-click symbolic execution. http://research.trust.salesforce.com/Introducing-Ponce-One-click-symbolic-execution/. Accessed Mar 2017
15. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
16. Kindsight Security Labs: Malware report - Q2 2012 (2012). http://resources.alcatel-lucent.com/?cid=177650. Accessed Mar 2017
17. RSA Security LLC: Current state of cybercrime (2016). https://www.rsa.com/content/dam/rsa/PDF/2016/05/2016-current-state-of-cybercrime.pdf. Accessed Mar 2017
18. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy, SP 2007, pp. 231–245 (2007)
19. Narayan, J., Shukla, S.K., Clancy, T.C.: A survey of automatic protocol reverse engineering tools. ACM Comput. Surv. **48**(3), 40:1–40:26 (2015)
20. Peng, F., Deng, Z., Zhang, X., Xu, D., Lin, Z., Su, Z.: X-force: force-executing binary programs for security applications. In: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC 2014, pp. 829–844. USENIX Association, Berkeley (2014)
21. Rodríguez-Gómez, R.A., Maciá-Fernández, G., García-Teodoro, P.: Survey and taxonomy of botnet research through life-cycle. ACM Comput. Surv. **45**(4), 45:1–45:33 (2013)
22. Saudel, F., Salwan, J.: Triton: a dynamic symbolic execution framework. In: Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, Rennes, France, pp. 31–54. SSTI, 3–5 June 2015
23. Schwartz, E.J., Avgerinos, T., Brumley, D.: 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 2010, pp. 317–331. IEEE Computer Society, Washington, DC (2010)
24. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2008 (2008)

25. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015 (2015)
26. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: SOK: (state of) the art of war: offensive techniques in binary analysis. IEEE Symposium on Security and Privacy, SP 2016, pp. 138–157 (2016)
27. Norman Solutions: Norman sandbox analyzer. http://download01.norman.no/product_sheets/eng/SandBox_analyzer.pdf. Accessed Mar 2017
28. Song, D., et al.: BitBlaze: a new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008). doi:10.1007/978-3-540-89862-7_1
29. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: 23nd Annual Network and Distributed System Security Symposium, NDSS 2016 (2016)
30. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: RAMBO: run-time packer analysis with multiple branch observation. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 186–206. Springer, Cham (2016). doi:10.1007/978-3-319-40667-1_10
31. Villeneuve, N., Sancho, D.: The "Lurid" downloader. Trend Micro Incorporated (2011). http://la.trendmicro.com/media/misc/lurid-downloader-enfal-report-en.pdf. Accessed Mar 2017
32. Wang, Z., Ming, J., Jia, C., Gao, D.: Linear obfuscation to combat symbolic execution. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 210–226. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23822-2_12
33. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Secur. Priv. **5**(2), 32–39 (2007)
34. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 732–744. ACM (2015)