A tool for analyzing and detecting malicious mobile code

Conference Paper · May 2006

DOI: 10.1145/1134285.1134426 · Source: DBLP

4 authors, including:

Akira Mori
National Institute of Advanced Industrial Science and Technology
40 PUBLICATIONS **255** CITATIONS
SEE PROFILE

Tomonori Izumida
Internet Initiative Japan
10 PUBLICATIONS **43** CITATIONS
SEE PROFILE

Toshimi Sawada
Kosakusha, Inc.
8 PUBLICATIONS **79** CITATIONS
SEE PROFILE

A Tool for Analyzing and Detecting Malicious Mobile Code

Akira Mori Tomonori Izumida
National Institute of Advanced Industrial Science and Technology (AIST)
Dai Building (10th Floor)
1-18-13 Soto-Kanda, Chiyoda-Ku
Tokyo, Japan
{a-mori, tomonori.izumida}@aist.go.jp

Toshimi Sawada Tadashi Inoue
SRA Key Technology Laboratory, Inc.
Marusho Bldg. (5th Floor)
3-12 Yotsuya, Shinjuku-ku
Tokyo, Japan
{sawada, now}@sra.co.jp

ABSTRACT

We present a tool for analysis and detection of malicious mobile code such as computer viruses and internet worms based on the combined use of code simulation, static code analysis, and OS execution emulation. Unlike traditional anti-virus methods, the tool directly inspects the code and identifies commonly found malicious behaviors such as mass mailing, self duplication, and registry overwrite without relying on "pattern files" that contain "signatures" of previously captured samples. The prohibited behaviors are defined separately as security policies at the level of API library function calls in a state-transition like language. The tool also features data flow analysis based on static single assignment forms, which are useful in tracing various values stored in registers and memory locations. The current tool targets at Win32 binary programs on Intel IA32 architectures and can detect most email viruses/worms that had spread in the wild in recent years.

Categories and Subject Descriptors:
D.4.6 [**Software**]: Security and Protection – invasive software; D.2.5 [**Software Engineering**]: Testing and Debugging – debugging aids, symbolic execution

General Terms: Security

Keywords: malicious code detection, static code analysis, code simulation, OS execution emulation

1. INTRODUCTION

We have been working to develop a technology to detect unknown computer viruses/worms [1]in Win32 executable file format that are activated on an IA32 processor [5]. Specifically, we are developing a tool to detect characteristic viral behavior, such as file infection and mass mailing, through analysis of API function calls. The analytical process consists of decryption by code simulation, static code analysis, and API function emulation. Defining anticipated virus behavior in the form of policies will allow detection of unknown viruses without relying on pattern definitions.

In the demonstration, we will present how this tool works, typical techniques to circumvent anti-virus software, and the ways the tool can respond to these techniques.

[1]The distinction between viruses and worms is not relevant here and only the term "virus" will be used for the rest of the document.

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

2. BACKGROUNDS

2.1 Conventional Anti-Virus Methods

Many commercially available anti-virus programs apply a detection system based on the "pattern (signature) matching" or "scanner" method. This system extracts certain binary code segments from known viruses, enters them into a database in the form of hexadecimal strings (called "patterns" or "signatures"), and matches files against this database to determine whether they are malicious or not. Generally, this system has the following disadvantages:

- The system cannot detect unknown viruses whose patterns are not contained in its database.
- It is difficult to create patterns that can uniquely characterize viruses and prevent safe files from being misidentified as malicious ones.
- Existing patterns are rendered inapplicable to matching simply with partial modification of the virus code as seen in numerous variants.

2.2 Self-encrypting and Polymorphic Viruses

To detect both known and unknown viruses effectively and accurately, we must be able to combat viruses that are capable of self-encryption and polymorphism. Self-encrypting and polymorphic viruses were originally devised to circumvent pattern-matching detection by preventing the virus generating a pattern. Unknown viruses applying this technique are even more difficult to detect.

A self-encrypting virus consists of an encrypted payload and code for decryption once in memory. Since the malicious part is encrypted, the behavior of the active virus cannot be determined by program code checking. Moreover, patterns can only be determined from the unencrypted segment (i.e., the decryption code), impeding pattern matching even further. As an enhanced version of the self-encrypting virus, a polymorphic virus was designed to avoid any fixed pattern.

This virus attempts to apply a different decryption code each time it is activated, through changes to its encryption method.

Unfortunately, most viruses today feature a mechanism for self-encryption and/or polymorphism, presenting a major problem in developing a precise detection system.

3. METHODS FOR DETECTING UNKNOWN MALICIOUS CODE

To develop a detection method for unknown malicious code, we must overcome a number of challenges, as follows.

Measures against self-encrypting and polymorphic viruses: There are two ways to detect such viruses when these are unrecognized: by cracking the encrypted code or by allowing the viruses to activate decryption on their own.

Methods of identifying malicious behavior: To determine whether an executable program will exhibit malicious behavior, it is necessary to analyze the special function calls (API function calls for Windows; system calls for UNIX) requested by the program from the operating system. Operating systems such as Windows and UNIX protect files and other resources from direct manipulation by ordinary programs. Even viruses must call functions from the operating system in order to perform any given malicious action. A mechanism is therefore required to identify sets of function calls that are indicative of malicious behavior.

For the first challenge, we decided to use code simulation technology to supplement static code analysis. This enabled us to analyze runtime behavior without executing virus code in a real machine environment. In the case of a self-encrypting virus, for example, the virus can decrypt its code by itself on the simulator; this code can then be used to perform static analysis.

As for the second challenge, we define anticipated viral behaviors as state machines where each transition is labeled with an API function with firing conditions on its arguments. Such descriptions are called detection "policies" and play a key role in increasing detection sensitivity to previously unreachable levels as they provide semantic information on the behavior of viruses. Detection is performed by driving policy state machines in the subsequent process of code analysis/emulation. The target program is identified as a virus when a final "denial" state is reached in any given policy.

In the following sections, we will detail our new detection tool and a method built on the principles outlined above.

3.1 Code Simulation

A code simulator precisely imitates changes in the internal structure of an IA32-architecture CPU (i.e., changes to registers, memory, flags, etc.) through the execution of machine instructions. The code simulator we use is operable both on Windows and Linux, and features basic debugging functions such as step execution, breakpoint setting, and memory dump.

As described above, we cannot analyze the behavior of self-encrypting and polymorphic viruses just by looking at program code. To track virus decryption and malicious behavior we must perform sequential simulation of executed instructions.

3.2 PE Loader Functions

Static code analysis for virus detection cannot be performed adequately using simulations of machine instructions alone. In addition to executable code, a program stores additional information – for example, externally defined functions and memory addresses (for storage of executable code or initial assignment of execution control) – in various locations within a file. The file must therefore be scanned to extract this information and to load the executable code in the proper memory addresses on the simulator. In other words, the simulator must perform the loader functions normally executed by the operating system. Since we focus on the Win32 environment, we built a program within the simulator to process PE (Portable Executable) binary format files, which are common in Win32 platform.

3.3 Processing of External API Function Calls

API functions are provided in an external library for application program use. To simulate code involving API function calls, a complete real machine environment must be prepared, which is in general too costly. Most API functions are unrelated to virus detection and the simulator must skip any instruction to call one of these unrelated API functions (instead recording the occurrence of a call), and proceed to a state in which this subroutine call is finished. To accomplish this effect, the following steps are required when identifying and processing subroutine calls corresponding to external function calls:

- Addresses must be identified in advance that are used to store the addresses of external functions determined at runtime. Specifically, the following addresses need to be identified, with their respective stored addresses and the names of the external functions:
  - An address allocated by the loader to each program upon loading; and
  - An address allocated by the virus code to itself at runtime, using an API function such as LoadLibrary (used to dynamically load a library) and GetProcAddress (to acquire the address of a function in the loaded library).
- Arguments are removed from the stack (according to the convention of Win32 dll functions).
- Return values are store in the EAX register.
- Policy checking is executed.

In practice, however, there are so many API functions that system extensibility may not be ensured if these steps are hard-coded in the simulator for each API function. As a solution, we designed the simulator to call a dummy function (called a "stub function"), instead of a real API function, to perform the necessary processing, and we prepared a separate library of stub functions corresponding to individual API functions.

Yet a standard Win32 environment has over a thousand runtime libraries, and a platform SDK includes an enormous number of API functions. It is thus extremely difficult to prepare stub functions for all of these libraries by filling argument byte counts, return values, and so on. We have developed a tool for generating stub function templates through mechanical processing of available Win32 system

information. Only additional emulation code (described below) needs to be filled manually.

3.4 OS Execution Emulation

To summarize, we can analyze programs and identify viral behaviors based on API function calls without requiring virtual execution of external library code if we have mechanisms: 1) to load executable files; 2) to load dummy library files filled with stub functions instead of real library files; 3) to remove arguments from the stack; 4) to set a return value; and 5) to check policies.

However, if we are to perform analysis in greater depth, we must collect more detailed information on the runtime environment, and we must generate in a virtual environment the side effects associated with program execution. In our tool, a virtual runtime environment is maintained for the first, and the partial emulation code may be included in the stub function for the second where it is necessary to return specific values depending on the execution contexts.

Specifically, the information about the following items must be maintained in a virtual runtime environment during static/dynamic code analysis:

- Registry
- Shell environment variables
- Heap areas: dynamic work areas in memory allocated by an API function such as "HeapAlloc"
- Files: directories and files created or opened by API functions such as "CreateFile"
- Exception interrupts
- Memory management including paging
- Thread management

Since the Windows registry itself is a large database and the uses of many of its registry keys are unclear, the virtual environment database only includes registry keys referenced by ordinary programs or likely to be abused by viruses.

Virtual heap areas are allocated to individual programs so that virtual memory areas are allocated in the virtual heap areas. Although no actual files are created, file structures are generated and managed based on the relevant stub functions, and the resultant information can be used in subsequent policy checking.

Exception interrupts can be handled by emulating SEH (Structured Exception Handling) mechanism, a Windows platform-specific mechanism. Paging is indispensable for basic access control. Our tool can handle basic paging as well as exception interrupts caused by the paging operation. In thread management, memory areas must handle thread-specific data known as TEBs (Thread Environment Blocks), and a multi-thread execution mechanism must also be available on the CPU simulator for synchronous and exclusive control. The tool supports open information with TEBs and simply analyzes elements of thread code in the order of generation with multi-thread. This is currently considered sufficient for behavior identification. However, if several threads coordinate to carry out a single malicious task, the tool may not be able to perform policy checks in high detail. One of our next goals is to enable the tool to emulate thread execution in greater detail.

3.5 Static Code Analysis

When simulating program code simply in a sequential manner, it is only possible to assess behavior that happens to occur during the simulation (as in the case of emulation-based dynamic protection). On the other hand, it is extremely difficult to check every element of code when a virus is capable of self-encryption or polymorphism.

In the early stages of development, we considered using a backtracking method for each branch instruction. However, taking snapshots of CPU usage and memory proved inefficient, and we found it extremely difficult to handle loop structures. The current control system first performs control flow analysis to read instructions for API function calls that must be checked, and then leads the simulator, on a priority basis, to execution paths that include these API function calls.

Besides control flow analysis to help simulations in later process, we have prototyped the data flow analysis function based on static single assignment forms [1, 6, 4]. Although the feature has not yet been fully integrated into the overall detection mechanism, it has been found extremely useful in analyzing new viruses that somehow escaped the detection. It helps identifying missing emulation capabilities of the tool and also discovering new hacking techniques.

3.6 Policy Checking System

Policies anticipate the unique, destructive behavior of virus. Specifically, they define scenarios such as mass mailing and file infection at the level of API function calls. Based on these policy definitions, the tool checks the results of code simulations, static code analysis, and OS emulation to determine whether defined types of behavior are identified within the code. Each policy is defined formally by state-transition machine in which transitions occurs when certain API functions are called under certain conditions.

The policy checking system drives the state-transitioning as defined in the policies to determine whether the "denial" state is reached. The tool is designed to activate the policy checking system through stub functions under the control of the simulator. If the simulator is forced to jump based on the static analysis results, it is led on a priority basis to the execution paths that include the API function calls to be checked.

4. POLICIES

Using the tool and method described in the previous sections, it is possible to analyze most existing viruses as well as ordinary application programs. The problem lies in determining which programs qualify as malicious; here it is essential to define the appropriate policies. Many policies are now implemented within this tool; the main ones are described below:

Mass email policy: Checks for the scenario in which the target program acquires an SMTP server address and random destination addresses based on registry information in order to issue a message. Simulation is performed using the SMTP protocol to determine whether mass emailing actually occurs.

Registry modification policy: Checks for any modifications to system settings, especially registry keys in which autostart programs are listed.

File modification policy: Checks for any modifications to write-protect directories/files.

File infection policy: Checks to determine whether the target program writes itself into files. Defines a scenario in which the program acquires a file handler by directory scanning, modifies it in memory, and writes it back to the file.

Process scan policy: Checks for a scenario in which the target program acquires process IDs from a list of processes under execution. This is defined as a policy because ordinary programs are unlikely to behave in this manner.

Self-code modification policy: Checks to determine whether the target program modifies headers, especially of import tables, in its memory image.

Anti-debugger/emulation policy: Checks for special behavior attempting to detect a debugger or an emulator.

Out of bounds execution (OBE) policy: Checks to determine whether the target program reaches the address space that are not declared in its header section.

External execution policy: Checks to determine whether the target program starts an external program with a CreateProcess or ShellExecute function call.

Illegal address call policy (IAC): Checks for any calls in which memory addresses were obtained by direct scanning of the memory image of library functions.

Self-duplication policy: Checks to determine whether the target program copies itself during execution.

Network connection policy: Checks to determine whether the target program performs FTP or HTTP communications. Here addresses are not checked as in the case of the mass email policy. Emulation of FTP and HTTP protocols is required.

These policies are written in a state-transition like language. They are transformed to C++ modules, compiled to runtime libraries, loaded onto the tool, and then called by stub functions to drive the state-transition machine.

5. EXPERIMENTAL RESULTS

Using the policies described in Section 4, we conducted detection experiments on approximately 600 virus/worm samples that have proliferated in recent years. Detection was successful with all of these viruses. We discovered that it is possible to detect not less than 95% of these samples using the IAC, OBE, self-duplication, and file modification policies. Even without the IAC and OBE policies, a detection rate of over 80% is possible using the file modification, self-duplication, and external execution policies. The false positives were kept minimum since the detection policy decides the false positive rate in a non-heuristic/algorithmic detection system like this.

More than 80% of the virus samples had not been analyzed prior to the experiments; these viruses were thus unknown to the tool. The tool was able to detect unknown viruses we have seen proliferate more recently, such as MyDoom, Bagle, Netsky, Mytob and all of their variants. These results confirm the effectiveness of this tool in real-world environments.

6. RELATED WORK AND CONCLUSION

Due to space limit, we only refer to most relevant papers here. An early binary detection method reported in [2] identifies API-level suspicious behaviors defined by security automata with help of control/data flow analysis. Christodorescu et al. proposed a robust instruction-level detection method based on formal semantics of binary code [3]. Both methods are static and do not inspect the encrypted part of malicious programs if there is any. Implementations reported in these papers are based on a commercial disassembler and the tool performance may not be enough for a real-world application yet. Our tool combines static methods with emulation of the CPU and the core operating system to demonstrate a reasonable detection performance. The tool is capable of both instruction-level and behavioral detection and has been tested on a large number of distinct samples captured on a working mail server for the last couple of years. Symantec Corporation has been granted a number of US patents including [8, 7] concerning emulation controls in detecting self-encrypting/polymorphic viruses. The method is heuristic and relies on artificial thresholds such as loop counts to determine when to stop emulation in a virtual execution environment. In our tool, control-flow analysis may invoke code simulation and there is no need for such thresholds.

Although the main function of the presented tool is malicious program detection, we found that it is also useful for reverse engineering purpose, such as checking library dependencies and program vulnerabilities. We are working to enhance the applicability of the tool and looking forward to presenting the results in this direction in the near future.

7. ACKNOWLEDGMENTS

We carried out this study from fiscal 2001 to 2003 under a commission from the Telecommunications Advancement Organization of Japan (currently NICT), and from fiscal 2004 to date from Japan Science and Technology Agency (JST). Our success in these efforts is due in large part to these supports and guidances.

8. REFERENCES

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[2] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable. In *Proceedings of the International Symposium on Requirements Engineering for Information Security SREIS'01*, pages 1–8, March 2001.

[3] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, May 2005.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[5] A. Mori. Detecting unknown computer viruses – a new approach –. *Lecture Notes in Computer Science*, 3233:226–241, 2004.

[6] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 12–27, January 1988.

[7] Symantec Corporation. Polymorphic virus detection module. *United States Patent*, 5,696,822, December 1997.

[8] Symantec Corporation. Dynamic heuristic method for detecting computer viruses using decryption exploration and evaluation phases. *United States Patent*, 6,357,008, March 2002.