Towards Automated Malware Creation: Code Generation and Code Integration

Andrea Cani
Independent scholar
Torino, ITALY
andreacani85to@gmail.com

Marco Gaudesi
Politecnico di Torino
Torino, ITALY
marco.gauesi@polito.it

Ernesto Sanchez
Politecnico di Torino
Corso Duca degli Abruzzi 24
ernesto.sanchez@polito.it

Giovanni Squillero
Politecnico di Torino
Corso Duca degli Abruzzi 24
giovanni.squillero@polito.it

Alberto Tonda
INRA, UMR 782 GMPA
Thiverval-Grignon, France
alberto.tonda@grignon.inra.fr

ABSTRACT

This short paper proposes two different ways for exploiting an evolutionary algorithm to devise malware: the former targeting heuristic-based anti-virus scanner; the latter optimizing a Trojan attack. An extended internal on the same the subject can be downloaded from `http://www.cad.polito.it/downloads/`

Categories and Subject Descriptors

I.2.m [**Computing Methodologies**]: Artificial Intelligence—*Miscellaneous*; D.1.2 [**Software**]: Programming Techniques—*Automatic Programming*

General Terms

Evolutionary Computation

Keywords

Malware, virus, evolutionary algorithms, security

1. AUTOMATED MALWARE CREATION

The analogies between computer malware and biological viruses are obvious: the very of an artificial ecosystem where malicious software can evolve and autonomously find new, more effective ways of attacking legitimate programs and damaging sensitive information is both terrifying and fascinating. *Malware* is a collective noun denoting programs that have a malicious intent – the neologism standing for *mal*-icious soft-*ware* [2]. Specifically, malware usually denotes hostile, intrusive, or simply annoying software programmed to gather sensitive information, gain access to private systems, or disrupt the legitimate computer operations in any other way. Since computer technology has nowadays emerged as a necessity in various aspects of our day to day life, including education, banking, communication, and entertainment, the threat posed by malware can't be overlooked.

This paper proposes to exploit μGP[1], a general-purpose EA toolkit [4], to create, or rather, optimize, malware. μGPis used with two different goals: to make malware undetectable by existing anti-virus program; to optimize the injection of the code inside a given host, creating a Trojan horse.

In *code generation*, μGPis used to create a new malware, with the precise intention not to be detected by existing scanners. While it would be theoretically possible to make μGPdiscover the patterns of malware autonomously, it is far more advantageous to feed the initial population with examples of working software to obtain successful individuals in a far more reasonable amount of time. Thus, the code of several malicious applications can be converted into μGP's internal representation of individuals. The evolution is then started, and μGPrearranges freely freely materials from the individuals provided in the initial population in order to create new malware.

The generated programs are checked to verify that its behavior is still compatible with that of malware, and eventually analyzed by the scan of a group of anti-virus software. Since the experience is focused on testing the static heuristics only, the chosen anti-virus programs perform the scan without relying on their database.

In *code integration*, μGPis used to determine the optimal position for hiding malicious code inside an existing executable. The goal is to perform the injection preserving both malware's and host's functionalities, and with no a-priori information about neither of them. μGPis used to efficiently explore the search space of possible blocks to replace, probing the target's code. The potential attacker is interested in finding vulnerable parts as large as possible, and it is important to notice that the search space for blocks of variable size inside a program quickly explodes, even for a relatively small target. Furthermore, finding potential vulnerabilities in compiled software is a task that would involve an intelligent analysis of the target program's behavior. Every individual in μGPrepresents a part of the program to be probed, and it is encoded as two integers: the first one (called $offset$) is the offset from the beginning of the compiled code, in bytes; the second (called $size$) is the size of

[1]Available under GPL from `http://ugp3.sf.net/`

SAC'14 March 24-28, 2014, Gyeongju, Korea.

http://dx.doi.org/10.1145/2554850.2555157

the part, again in bytes.

The tool distinguishes between two types of areas of potential interest from an attacker's point of view: *Type I* areas represent blocks of code that almost always skipped during a regular execution, like branches after a flow control instructions that are rarely activated; *Type II* areas are usually not processed by the normal flow, and often appear after the end of the main function of the program, like functions that are infrequently called. The rationale is to use a *Type I* area to inject a vehicle, overwriting the branch with a call – as few as 22 bytes are sufficient to save all registry values, call the malware code, and restore the original values. Then, store the actual malware into one ore more *Type II* areas. The tool does not need any hint, and it is is able to autonomously discriminate between the two types of area observing the behavior of the program after integration.

2. EXPERIMENTAL RESULTS

For the experimental evaluation, the code of the virus *Timid* is inserted into the initial population. Timid is a relatively simple malware, a file infector virus that does not become memory resident, firstly discovered in 1991 and rumored to be an escaped research virus [1]. Each time a file infected with Timid is executed, it copies its own code into other uninfected .COM files in the same directory; if there are no more .COM uninfected files, a system hang occurs. Timid is chosen for the experiment because of several desirable characteristics: despite its age, it still works on the operative system; its code is available in the public domain; and its behavior is very predictable and controllable. Thus, checking if the modifications of Timid created by the evolutionary framework still behave as the original malware becomes a relatively straightforward process.

A set of 5 .COM files, taken from those available in the directory `DRIVE:\Windows\system32\`, are used as a test for the infection capability of an individual, their integrity being checked with a md5 cryptographic hash function [3]. An ensemble of 4 different freeware anti-virus applications is selected to verify the ability to escape detection. Each experiment, run on a Notebook Intel Centrino running Windows XP Service Pack 2, takes 10 to 15 minutes to complete. Starting from the same initial population, containing only one individual modeling the original code of Timid, 100 runs of the framework are executed. The successful experiments terminated very quickly, in an average of 6 generations, with a standard deviation of 2.5. It is worth noticing that the best individual in generation 1 is already able to deceive the heuristics of two of the anti-virus applications; nevertheless, the fitness of the best individual progresses steadily during the generations.

The code integrtion approach was tested on two target executables: `SPLIT.EXE`[2], a small program (46,6 kB) able to split files of any kind into smaller parts or rebuild the original; `TESTDISK.EXE`[3] (315.2 kB), an open-source data recovery software available for different platforms.

Each experiment, run on the same notebook, takes about 30 minutes to complete. For `SPLIT.EXE`, μGPfound 1 zone of Type I and 32 zones of Type II, ranging from 65 to 1511 bytes, thus showing potentially vulnerable positions for an attack. `TESTDISK` proves slightly more resilient to attacks:

SPLIT.EXE	
offset interval	(0,43000)
Evaluations	300
Type I (zones found)	1
Type I (largest)	334
Type II (zones found)	32
Type II (largest)	1,511

TESTDISK.EXE			
offset interval	(0,43000)	(0,10000)	(0,2000)
Evaluations	15,000	2,000	300
Type I (zones found)	-	1	1
Type I (largest)	-	33	25
Type II (zones found)	3	4	3
Type II (largest)	179	167	183

Table 1: Summary of the experiments for code injections. While SPLIT.EXE shows vulnerabilities even after a first run, several attempts are needed to find exploitable areas in TESTDISK.EXE

the first attack only uncovers 3 small zones of Type II, ranging from 12 to 179 byte. Since the zones appear to be concentrated in the first part of the executable, subsequent attempts are made, progressively restricting the interested area twice, and for both settings, a zone of Type I if eventually detected. See 1 for details.

3. CONCLUSIONS

The short paper shows how EAs can be exploited in anti-malware research. While the research is still at an early stage of development, its potential is apparent: producing new malware with negligible human intervention could be extremely advantageous to anti-virus producers to test and enhance their products. Moreover, the creation of trojans could also be used, for example, to test the security of computer infrastructures.

Acknowledgments

The authors would like to thank Muddassar Farooq, Ralph Roth, and Waqar Ali for their availability and insightful discussions.

4. REFERENCES

[1] McAfee. Threat center: Timid entry. http://vil.nai.com/vil/content/v_1240.htm, 1991.

[2] R. Moir. Defining malware: Faq. http://technet.microsoft.com/en-us/library/dd632948.aspx, October 2003.

[3] R. Rivest. The md5 message-digest algorithm. *RFC 1321*, 1992.

[4] E. Sanchez, M. Schillaci, and G. Squillero. *Evolutionary Optimization: the μGP toolkit*. Springer, 1st edition, July 2012.

[2]See `http://www.iacosoft.com/home/split.txt`

[3]See `http://www.cgsecurity.org/wiki/TestDisk`