Thwarting Attacks in Malcode-Bearing Documents by Altering Data Sector Values

Wei-Jen Li and Salvatore J. Stolfo
Columbia University
September 9, 2008

ABSTRACT

Embedding malcode within documents provides a convenient means of attacking systems. Such attacks can be very targeted and difficult to detect to stop due to the multitude of document-exchange vectors and the vulnerabilities in modern document processing applications. Detecting malcode embedded in a document is difficult owing to the complexity of modern document formats that provide ample opportunity to embed code in a myriad of ways. We focus on Microsoft Word documents as malcode carriers as a case study in this paper. To detect stealthy embedded malcode in documents, we develop an *arbitrary data transformation* technique that changes the value of data segments in documents in such a way as to purposely damage any hidden malcode that may be embedded in those sections. Consequently, the embedded malcode will not only fail but also introduce a system exception that would be easily detected. The method is intended to be applied in a safe sandbox, the transformation is reversible after testing a document, and does not require any learning phase. The method depends upon knowledge of the structure of the document binary format to parse a document and identify the specific sectors to which the method can be safely applied for malcode detection. The method can be implemented in MS Word as a security feature to enhance the safety of Word documents.

1. Introduction

Modern document formats are fundamentally object containers that provide a convenient "code-injection platform." One can embed many types of objects into a document, not only scripts, tables, and media, but also arbitrary code used to render some embedded object of any type. Many cases have been reported where malcode has been embedded in documents (e.g., PDF, Word, Excel, and PowerPoint [10,16,20]) transforming them into a vehicle for host intrusions. Malcode bearing documents can be easily delivered and bypass network firewalls and intrusion detection systems when posted on an arbitrary website as a passive "drive by" Trojan, transmitted over emails, or introduced to systems by storage media such as CD-ROMs and USB drives. Furthermore, attackers can use such documents as a stepping stone to reach other systems, unreachable via the regular network. Consequently, any machine inside an organization with the ability to open a document can become the spreading point for the malcode to reach any host within that organization.

In this study, we focus on Microsoft Word document files. There is nothing new about the presence of viruses in email streams, embedded as attached documents, nor is the use of malicious macros a new threat. Word macro viruses have been well studied[1] [5,31]. In this paper, however, we focus on another attack type, the exploit of Word vulnerabilities where malcode is embedded in normal-appearing document data sectors. Microsoft Office documents are implemented in Object Linking and Embedding

[1] To thwart malicious macros, the solution provided by Microsoft is to disable all macros that do not contain authorized digital signatures. While malicious macro problem is not completely solved, attackers can embed malcode using stealth strategies that are harder to detect.

(OLE) structured storage format, in which any arbitrary code could be embedded and executed and various vulnerabilities would be exploited. For example, Word may harbor various memory corruption exploits such as buffer overflows [43], integer overflows [35], format string overflows [37], and vulnerabilities with other applications (e.g., HP Storage Management Appliance [36] and WMF vulnerability in MS graphic rendering engine [42]). Adversarues may craft data that cause a buffer overflow which directs the program to an invalid memory address. Consequently, Word jumps to a particular location to execute arbitrary code. Combined with polymorphic [9,18,39,44] and mimicry [19,25,32] strategies, the embedded malcode is difficult to detect by IDSes and currently no easy solution is available other than not using Word altogether. However, Adobe PDF is not immune to the same attack strategies.

To counter these stealthy attack techniques, we developed a malcode detection technique we call the Arbitrary Data Transformation (ADT). The strategy is to modify and damage possible embedded malcode by arbitrarily changing the data values of certain content portions of Word files. Having modified data processed by MS Word, execution of those portions with embedded malcode will very likely be damaged causing a "system crash" that is easily noticed. On the other hand, altering normal data used to describe the content of the documents would not crash the program in cases where no malcode existed. The display of the document might be different or distorted but Word won't crash. For example, the characters, font size, and type are changed, but Word can still display the document without causing exceptions.

The technique is intended as a safety check when opening Word documents. ADT might be applied within Word in a sandbox or virtual machine, or by a separate analysis application such as SPARSE [22,23] or a third party AV scanner, and if the document is deemed benign, the document would be opened in its normal fashion after reversing the ADT transformation. Hence, ADT is a reversible transformation and detection process applied to Word documents in a safe environment.

The ADT strategy was inspired by the technique of instruction set randomization (ISR) [3,14] and address space layout protection (ASLR) [4,6] for thwarting code injection attacks. ISR randomly maps the instruction set of a program and requires a key to decode the instructions. The attacker cannot be certain what the real code is without a key to decode it. ASLR changes the memory layout so the malcode cannot know where their targets are located in memory. The malcode is thus uncertain about the location of specific code it seeks to target.

ADT is different than ASLR. It does not protect programs from exploitation of some vulnerability that transfers control to the modified sections. On the other hand, we let the embedded code be triggered. Our ADT strategy arbitrarily changes data values in the document that would damage any embedded malcode which would be noticed when the exploit triggering the malcode is executed. ADT is different than ISR. ADT doesn't need a key to decode the transformed data values. We open the "transformed" document using MS Word and let the application either fail or not, essentially *using MS Word as detector*.

Each single byte has 256 possible values, and an n-byte data sequence has 256^n possible permutations. As long as the changed byte values are arbitrarily chosen in a random non-predictable fashion, the attacker cannot predict or guess what transform we may apply to document's data sections. Since the ADT process can be performed every time when a document is opened, the changed values can be different each time a document is opened. Therefore, it is impossible to continuously guess the key by using any brute force strategy [27], which has been shown can compromise the ASLR and ISR protection mechanisms. Each attempted probe requiring opening the document produces a new transformed instance of the document, creating essentially moving targets thwarting brute force guessing strategies.

The ADT method is similar to *emulating* suspect content. For example, modern AV systems run an emulator to see if certain content being processed is likely code embedded in data that should not have code. The ADT strategy attempts to detect code by damaging it and noting a system failure. A third party AV vendor would be able to apply emulation inside documents but that process may be expensive and requires that the document format be parsed to identify data sections where emulation would be applied.

Given these constraints, the document application vendor is in the best position to perform the check for hidden malware within the application directly. Sections parsed in the proprietary Word format might be directly tested using the ADT strategy without the need to develop an emulator.

The rest of the paper is organized as follows. In Section 2, we discuss related work and the prior work focusing on malicious document detection. We detail the ADT technique in section 3 and evaluate our approach in Section 4. Finally, we conclude the paper in Section 5 after discussing future work to improve upon the method.

2. Related Work

2.1 State-of-the-Art Detection Techniques

Approaches to malware detection can generally be grouped into two areas: the static and the dynamic approaches. Static techniques analyze the binary content of files that may harbor malware without executing the code, while dynamic detection systems execute the code in emulated environments and observe the run-time behavior to determine malfeasance.

Traditional development of virus signatures required human experts to manually analyze the program and extract specific byte contents as signatures [15,31]; this was a laborious, time consuming procedure. The later improved methods are algorithmic scanning that look for frequent common patterns in the binary content [17,28] and heuristic analyses that search for suspicious code or behavior at specific locations in memory or files [1]. In addition, n-gram modeling techniques and statistical anomaly detection have been applied effectively to detect malware [22,30,33]. However, none of these approaches can guarantee detect zero-day attacks will be detected without false positives.

Since it is difficult and time consuming to analyze the intent of malicious software by manually disassembling the code or by using reverse engineering methods, dynamically executing the code and observing the system behavior has become a common method for malware detection. Numerous papers have been published on this topic [11,12,31,47,48].

2.2 Steganalysis, Polymorphism, and Mimicry Attacks

Recently, steganalysis, polymorphic, and mimicry techniques have been used counter and evade the static and dynamic detection techniques. Steganography is a technique that hides secret messages embedded in otherwise normal appearing objects or communication channels. Provos [44] studies cleverly embedded “foreign” material within media objects that evades statistical analysis while maintaining what otherwise appears to be completely normal-appearing objects (e.g., a sensible image). The general class of the steganographic embedding of secret messages may be viewed as a “mimicry” attack, whereby the messages are embedded in such a fashion as to mimic the statistical characteristics of the objects in which the messages are embedded. Malcode embedded in documents may be considered a form of steganography, but here with the aim of code execution.

Similarly, polymorphic techniques have been exploited to deceive signature-based IDSes. ADMutate [39] and CLET [9] craft polymorphic worms with vulnerability-exploiting shellcode to defeat simple static anomaly detectors. According to the statistical distribution learned by sniffing the environment, Lee et al. [18] inject morphed padding bytes into the code allowing the code to have a “normal” appearing statistical characterization. Other researchers suggest that systematically injecting noise or fake samples into the training pool can obstruct the learning algorithms from generating reliable signatures [24,26]. As a result, all detection algorithms based on learning a static pattern would be evaded by adding patterns that are similar to the exploit’s invariants. Song et al. [29] suggest it is futile to compute a set of signatures of malicious code, and hence identifying malcode embedded in a document using signature-based approaches may not be the wisest strategy.

The ADT method described in this paper can counter these obfuscation techniques since either the encrypted code or the decoder that is embedded inside a document is arbitrarily modified and its functionality will be damaged and very likely detected if the modification results in a system crash.

2.3 Code Injection Attacks and Detection Techniques

Various defense techniques have been developed to protect systems from code injection attacks. Some concentrate on preventing buffer overflow by protecting the return address, specific data values, and pointers from being overwritten [7,8,21]. However, all of these techniques only detect some specific types of exploits (e.g., overwritten of return address), while others can still bypass the protection.

Obfuscation and randomization are approaches that try to solve the code injection problem more generally. They are not protection mechanism; however, randomization techniques make the exploits more difficult to succeed. Instruction set randomization (ISR) defuse code injection attacks by randomly changing the instructions used on a host machine or application [3,14]. ISR prevents the execution of embedded malcode by randomizing the relationship between the op code and the instructions for programs. The code section of an executable is encrypted with a key, which is stored in memory, and an instruction is decrypted when it is loaded to the processor. The drawback however is that the emulator design requires extra memory to handle shared libraries and ISR cannot disable mimicry attacks that use legal instructions.

Address space layout randomization (ASLR) [4,6] that randomizes the base address of stack, heap, code, and data. As a result, the embedded malcode jumps to a wrong location or pointers won't be able to locate the malcode it tends to launch. Shacham et al. [27] studied the effectiveness of address randomization. They implemented a brute force attack (they called a derandomization attack) that repeatedly probes the target and can compromise a 16-bit key space address randomization protected machine within a few minutes. They stated that runtime address-space randomization techniques are not as effective as once commonly believed, and any buffer-overflow attack could be crafted to work against address randomization.

Jiang et al. [13] developed a system named RandSys that combines the ISR and ASLR techniques that can effectively thwart a wide variety of code injection attacks with a small overhead. There proposed system greatly enhances protection because it is more difficult to correctly guess the combined keys.

2.4. SPARSE: A Hybrid System to Detect Malicious Documents

We previously introduce the SPARSE system [22,23] developed for malicious document detection. SPARSE is a hybrid system that combines multiple detectors and various detection strategies to detect malicious documents as well the location of the embedded malcode with in the document.

The static detector built into SPARSE is based on the Anagram algorithm [33] that characterizes the binary contents of document files. The system includes a fine-grained static parsing mechanism to separate different types of objects embedded in the documents to model them separately, avoiding mixing their statistics together. The detailed algorithm used to detect malicious documents is described in [22], in which a 2-class mutual information modeling technique (i.e. one benign model and one malicious model) is suggested.

SPARSE has a second dynamic event detector system that models the system's run-time behavior such as file creation/modification, module loading, registry access/change, and process activity. In addition, the system includes a mechanism that activates and examins passive embedded objects that require human action to launch. Thus, the dynamic detector has an automaton that automatically interacts with the

SPARSE also has a mechanism to locate the malicious sector harboring the embedded malcode in a document by selectively removing suspicious sectors one at a time and testing the rest of the document for identifying the presence of the malcode. This strategy not only locates the malcode but also uses the

extracted sectors containing malcode in an integrated feedback loop to update the models and to improve the accuracy of the detectors over time.

In our earlier study [23], we found combining multiple detectors and using optimized strategies, the hybrid SPRASE system could achieve high detection accuracy with a low false positive rate, and the experimental result of the hybrid system was superior to the result of using any individual detector alone.

As an additional dynamic test procedure integrated in SPARSE, ADT is designed to detect stealthy embedded attack that may use mimicry strategies that can evade both static and dynamic detection approaches. ADT does not need any training model; instead, it disables potentially embedded malcode by altering vulnerable data values. The detailed technique is described in the next section.

3. Arbitrary Data Transformation

Before detailing the approach, we briefly describe the attacks we address. The attacks we consider are not traditional macro viruses, which are typically VBA code and are usually located in the "Macros" or "WordDocument" sector[2]. Figure 1 displays a parsed document in the OLE structural format, which contains nodes and directories; we call each node a "sector." In sectors such as "1Table," Word may harbor various exploits such as buffer overflows [35,36,43,66] or vulnerabilities with other applications [42]. The attackers may craft data that exploit the vulnerabilities which redirect the execution of Word to a particular location to execute arbitrary embedded malcode.

Another type of attack strategy is to embed malcode in the padding areas of the binary file format (the space filled with 0 byte values to align blocks of code on disk for fast disk access) of documents or to replace normal textual data with malcode. This stealthy multi-partite strategy[3] [30, 34] can lie dormant in the file store of the target environment awaiting a future attack that would extract and activate it. In contrast, normal documents do not contain executable code in these areas. Further, the embedded code can be shaped to appear as if it were normal text through "spectrum shaping" [45,46].

We studied the Microsoft Office Binary File Format Documentation [41, 67] to understand and analyze the Microsoft document format to parse the binary into the correct sectors. The proprietary Microsoft documentation contains a very substantial list of sectors with intricate structure. In general terms, we may categorize the byte content of a Word document into two types: the data and the pointers to the data (i.e. the binary contents that indicate the offset or the length of data). When processing a document, Word first looks for the pointers, which are at either absolute or relative locations; these pointers tell Word where to find the data that specify the document rendering function by its type (e.g., text, images, or tables). The data values are either exact data values (e.g., images) or information telling Word how to display the data, such as objects' sizes and styles (e.g., data structure variables).

Specifically, many pointers are contained in the File Information Block (FIB) that are embedded in the beginning of the WordDocument sector, and these pointers indicate either the offset or the length of data embedded in sectors such as the other portions in WordDocument or the 1Table sector. The rest of the WordDocument sector contains the text, macros, and some other data. The 1Table sector, which is currently the most vulnerable portion of Word documents, contains approximately 130 data structures including many types of pointers and data. The most prevalent attacks in the wild have been primarily stealthily embedded attacks in the 1Table sector. For this study we obtained a number of Word documents

[2] In general, the OLE structure is like a small file system that are composed with various "sectors" such as WordDocument, Macros, and 1Table. More details about the Word document structure can be found in other works on malicious document detection [32,33].

[3] An earlier work [47] demonstrated that embedding malcode, even known viruses, in the padding areas could evade detection by COTS AV scanners.

attacks which were all embedded in the 1Table sector. However, our detection strategy is not limited to only 1Table but can also be applied to the other sectors with a correctly implemented parser.

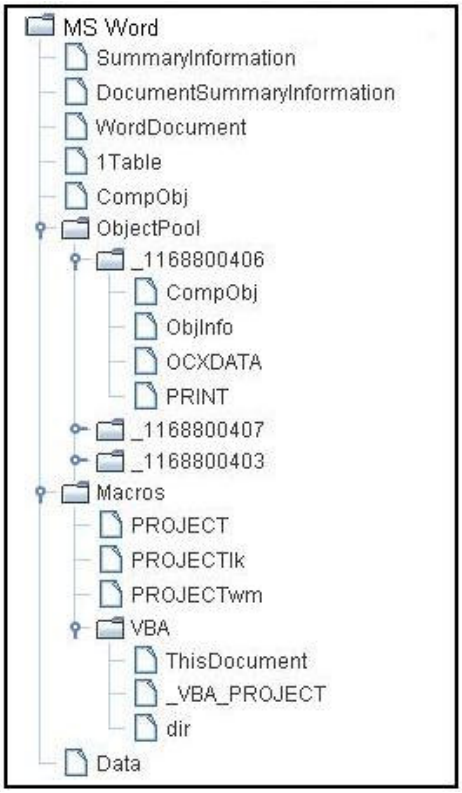

Figure 2: An example of the internal structure of a parsed document in OLE format

The data in 1Table can be text such as "Times New Roman," identifying fonts, or numbers used to define sizes, locations, flags and indices, or *code blocks* that can be script or encrypted code. In addition to the pointers and data, there are some "magic numbers" in some cases which are keywords or special values that cannot be changed without crashing Word, for instance, the magic numbers indicating the beginning of a document or a sector, streams of Microsoft Office reference schemas [40], and some specific numbers used to indicate the end of structures. All of these values cannot be arbitrarily modified.

Embedding malcode in pointers is difficult. First, if the malcode is crafted as a pointer value, the attacker has to ensure that the document doesn't break Word before the malcode is launched, i.e. arbitrarily replacing a pointer value with another would likely crash Word when it processes that pointer. Second, since the pointers are usually short, i.e. from 1 bit to a few bytes, and not contiguous, there is not sufficient space to place the malcode in a pointer area without overwriting the data or padding values adjacent to the pointers. Finally, it is hard to exploit these pointers without being fairly easily noticed, e.g., by introducing high entropy values in a certain area, significantly increasing the document size, or crashing the system. As a result, an attacker would likely find it convenient and safe to embed their malcode in the data sections or the padding areas. For example, the data that cause buffer overflow can be any arbitrary value; some are embedded in the data areas while others overwrite some pointers. However, the executable malcode or the decoder of the encrypted malcode are always embedded in the data sections or padding areas, for this is far easier to craft without introducing errors in the documents.

Based on the above observation of the attacks we studied and MS Word format's complexity, in order to counter this type of attack our goal to detect malcode is to alter the byte values of the malcode that may be embedded in the data or padding areas. To this end, we arbitrarily change the data portions to arbitrary chosen different values, for all of the non-zero data values that can be safely changed. As a consequence,

the ADT strategy will change the binary content of illegally embedded code to arbitrary values and will force it to crash or to be disabled. That is, ADT does not avoid the execution of buffer overflow or other vulnerabilities but damages the malcode to purposely crash. On the other hand, normal data used to describe the documents should be able to be changed into other values without serious error; the display of the document may be changed even appearing incomprehensible, but the system won't crash when malcode is not present. To understand which values can be changed, we studied and analyzed the (proprietary) Microsoft documentation[4] [41] and manually analyzed thousands of Word documents. For example, each of the 130 structures in the 1Table sector used to describe the documents structure may contain several types of data, including pointers, data, and keywords. We analyzed each of the structures and observed which data could or could not be safely changed.

For all of the byte values that can be changed (i.e. neither keywords nor pointers), we increase or decrease those data values by some arbitrarily chosen displacement x (e.g., changing the character "A" to "Q"). In our test cases, the value of x ranged from 1 to 3 (or -3 to -1), so a y-byte long data section has $\left|2x^{y}\right|$ (x can be negative) possible permutations. The range of x is 256, which is the total possible values of a byte. In this paper, we arbitrarily changed the values with a displacement from 1 to 3 to provide a proof-of-concept demonstration. As future work, the value displacements may be changed to specific values defined according to the type of the structure using that data so the display of documents won't be damaged but just transformed into another style or type. Thus, the range of the displacement value can be far wider than what we used in this study. Moreover, as we noted ADT does not need to convert the transformed values back as in ISR to render a document. Word will render the transformed data segments. As a result, displacement value x, can be different every time when we perform the transformation, and we may arbitrarily apply different displacement values to each byte value if we choose. Hence, the brute force attacks [27], which keep probing to learn the correct encryption key, will not able to successfully guess "the key".

In many cases, as we noted, the display of the document will likely be distorted after applying ADT, as we may expect. For example, the font display for ASCII data used in the 1Table sector may appear with the "Times New Roman" font name, where the corresponding byte values are "54 69 6D 65 73 20 4E 65 77 20 52 6F 6D 61 6E." These data values are the data that describe the text type font. Arbitrarily changing any of these values to another value, including the extended ASCII characters, would not cause Word to crash. Word would choose a default font or style if the transform was incomprehensible to Word.

Figure 2 compares a Word document before and after the ADT process. In this figure, the data representing the text type (i.e. FFN, Font Family Name) are arbitrarily changed, but the document can still be opened and displayed without any error. The worst outcome of ADT is that Word displays a blank page or displays some strange characters when malcode is not embedded in the document.

[4] We signed a non-disclosure agreement and hence cannot reveal the format details in a public document. Interested readers can request the documentation from Microsoft: officeff@microsoft.com

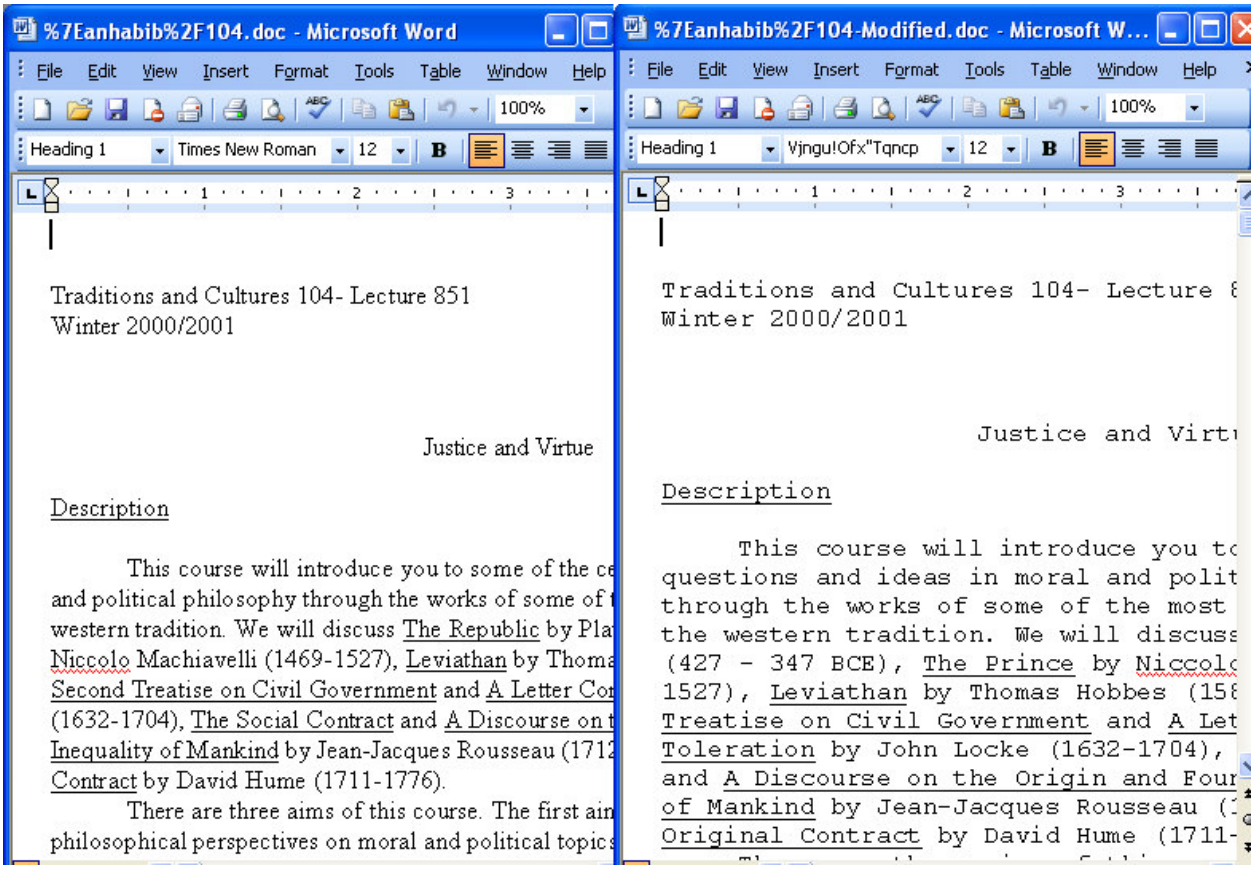

Figure 2: Before (a) and after (b) transforming the data that used to render the text type. In (a), the text type is "Times New Roman." In (b), both the characters representing the text type and the reference index have been changed.

On the other hand, stealthy embedded malcode residing in the data portion, if there is any, will also be changed, and subsequently either Word will crash or the malcode will be disabled when an attempt is made to execute it. For example, the hexadecimal Opcode value "6A" and "EB" represent the push and jmp x86 instructions, respectively. If the byte values are increased by 1, they become "6B" and "EC" which are not correct Opcodes. Even though sometimes the changed code is valid, it can become another completely unintended instruction. As a result, the program or the OS will not be able to correctly execute the attackers' shellcode and will either crash or terminate the process. Figure 3 and 4 display the binary content of two buffer overflow attacks. In Figure 3, the list of 4-byte values is a list of pointers (i.e. the fcPlfLfo structure), and the zeros are padding values. The highlighted values are embedded shellcode which is an infinite loop causing memory corruption. In Figure 4, the structure is used to describe the Font Family Name (FFN). ADT does not change the values of the pointers; however, the byte values of the malcode embedded in the padding area will be changed after the ADT process. In any case, the attack will be disabled. Hence, whether the shellcode exhibits obvious or mimicry behavior, our system can detect it by the ADT process.

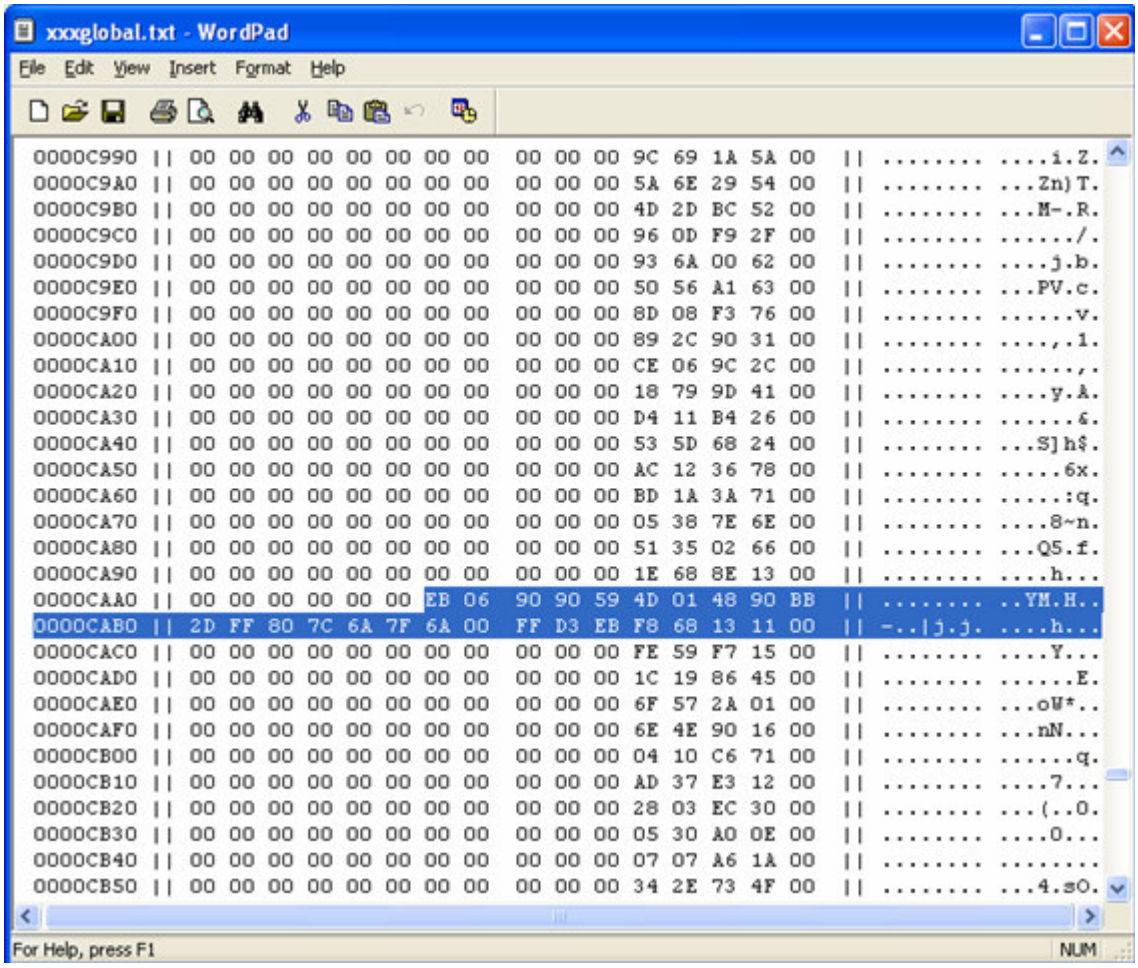

Figure 3: The binary content of malicious shellcode embedded in a Word document (fcPlfLfo).

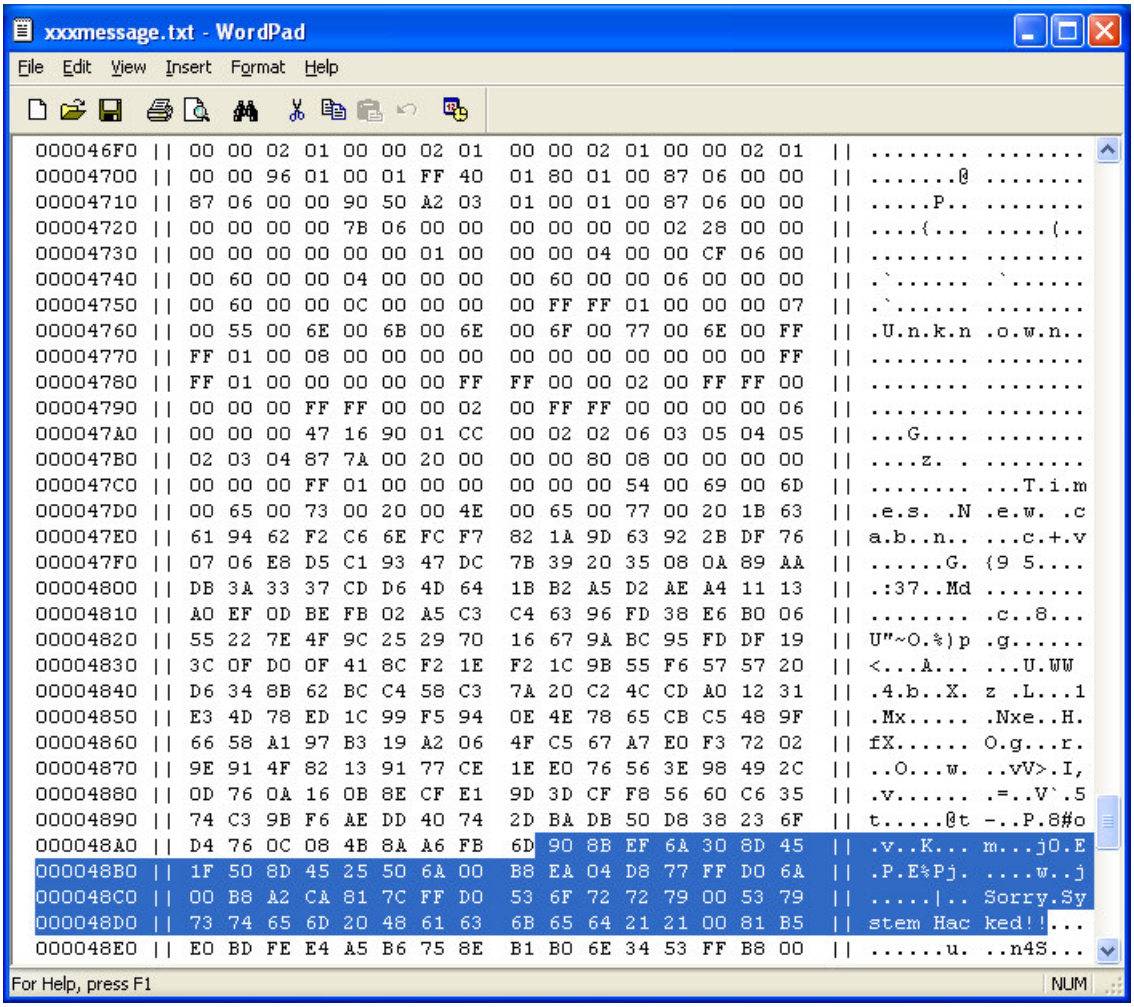

Figure 4: The binary content of a malicious shellcode embedded in a Word document (fcSttbfffn).

Indeed, some changes may cause Word to be incapable of displaying the content at all, and these documents are considered as false positives in our experiments, which will be presented in the next section. However, Word does not crash in these cases; instead, it displays a message indicating that the document may be damaged. Two examples are shown in Figure 5 and Figure 6. These error messages vary depending on the version of Word or patch level. We doubt a true attack would cause this message to be displayed. If the attack was crafted to display this error message (but not crash the system), the attack would announce itself to the user anyway! Furthermore, it is possible that attackers can replace the "non-changeable values," such as the keywords, by crafted malcode; however, this is much harder than embedding malcode in the data area because the non-changeable values are usually short, not contiguous, or special keywords.

By way of summary, the ADT technique will disable or corrupt the stealthy embedded malcode, but not normal data. Thus, if a document behaves normally before the ADT process and crashes afterwards, we consider that it contains stealthy malcode (malicious); if it doesn't crash after the process, it is deemed benign.

Figure 5: An error message when opening a damaged document.

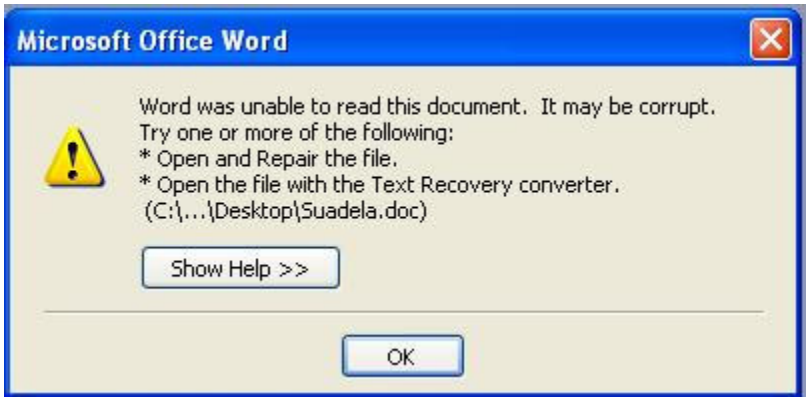

Figure 6: An error message when opening a damaged document.

4. Evaluation

To evaluate the ADT strategy, we tested 15 "1Table" attacks supplied to us by an external party, where we were able to manually verify the accuracy of our method. These attacks include memory corruption (i.e. stack buffer overflow) and Trojan-Droppers, exercised vulnerabilities known to exist in earlier versions of Word that have since been patched for the most recent versions. For each of the tested documents, we automatically applied ADT, and subsequently we observed and compared the system behavior (i.e. whether there is any exception or error) when executing both the original documents and their modified versions in a virtual machine.

The summary results displayed in Table 1 show 11 out of the 15 attacks were successfully detected – the data modification forced system errors. For the memory corruption attacks, Word couldn't display the documents, instead, it either showed a "Your system is low on virtual memory" or ran for a long time until the system crashed. A sample of code is shown in Table 2. This shellcode, which was an infinite loop to corrupt memory, was embedded in the data area, and changing any of the byte values, except "01" and "02," would disable the function (i.e. the code would become meaningless). After applying ADT to these malicious documents, we forced Word to terminate immediately[5] when opening the documents. It appeared Word found incorrect embedded code and stopped the process. For the other attacks, the situations were similar; Word was either terminated immediately or displayed an error message.

[5] This was tested in Windows XP SP2 and Word 2002 with no patches. Other versions of Windows and Word may exhibit different error handling mechanism.

Table 1: Summary of the tested known attacks. The original file names are not revealed because of confidentiality concerns. Three Trojan-Droppers contained code in multiple structures. The numbers in the parentheses are the number of suspicious structures. The structure names are defined by the Microsoft documentation [41].

Attack Index	Attack Type	Result	Location
1	Memory corruption	Detected	FIB
2	Memory corruption	Detected	fcPlfLfo
3	Memory corruption	Detected	fcPlfLfo
4	Display pop-up message	Detected	fcSttbfffn
5	Encrypted malcode	Detected	fcSttbfffn
6	Steganographic	Detected	fcSttbfffn
7	Opened system32/calc.exe	Detected	fcPlfLfo
8	Program error	Detected	fcSttbfffn
9	Program error	Detected	fcDop
10	Trojan-Dropper	Detected	fcSttbfffn
11	Trojan-Dropper	Detected	fcSttbfffn
12	Trojan-Dropper	Disabled	Multiple (5)
13	Trojan-Dropper	Disabled	Multiple (25)
14	Trojan-Dropper	Disabled	Multiple (22)
15	Trojan-Dropper (an attack used to against Tibet support organizations in March 2008)	Couldn't change the values	fcSttbfBkmkFactoid

Table 2: The shellcode example of a memory corruption attack.

Byte Value	Code	Comment
BB XX XX XX XX	mov ebx, XX XX XX XX	XX XX XX XX is the attack
6A 01	push 01	Add an argument
6A 02	push 02	Add an argument
FF D3	call ebx	Call the attack
EB F8	jmp F8	Jump 8 bytes backward, which is 6A 01

After applying ADT, three Trojan-Droppers did not introduce system errors; instead, their malicious behaviors, creating some registry keys and files, was disabled. This was what we expected: the transformed malcode would either be disabled or introduce program error. Nevertheless, we labeled them as false negatives because we couldn't automatically verify whether the disabled events were benign or malicious by simply observing a system crash. Other detectors could be used in this case to make a final informred decision. Furthermore, we couldn't transform one of the Trojan dropper's malcode (the 15th attack shown in Table 1), which was a recent attack that compromised a few targeted organizations [38] related to the recent Tibet/Chinese government dispute. According to the official Microsoft Documentation [41], this malcode was embedded in an *undocumented structure*. Being an undocumented

structure, we couldn't transform it without introducing a significant number of false positives. However, it is very interesting that the attackers had knowledge to embed malcode in this structure[6]; we didn't since it was undocumented and did not appear in the MS Word format specification.

To establish a False Positive rate, we also performed the same test scenario on 1516 benign documents. Most of them behaved normally after the ADT process – they did not cause the system or Word to crash after applying ADT; however, we could not apply the strategy to some complex benign documents.

Among these 1516 test documents, 9 caused Word to display an error message after applying the ADT process (e.g., a pop-up window displayed "Word was unable to read this document. It may be corrupt."). Thus, we found some particular documents with distinct sectors that cannot be arbitrarily changed; for these we require deeper knowledge of Word, as in the case of the undocumented sectors described above. For this study, although we understood the names of the structures and fields, we couldn't obtain complete knowledge such as the meaning of some field values. For most of the normal documents, many of the particular strings could be arbitrarily changed without introducing error. However, in a few cases, the document couldn't be displayed after applying ADT on some subset of these strings. Table 3 summarizes the data that caused false positives.

Table 3: The shellcode example of a memory corruption attack.

Structure name	Description	Comment
fcSttbfffn	Table of font name strings	Some strings cannot be arbitrarily changed.
fcPlfLfo	List Format Override	Some flag values cannot be arbitrarily changed.
fcSttbfbkmk	Table of bookmark name strings	Some strings cannot be arbitrarily changed.
fcSttbttmbd	True type font embedding string table	Some strings cannot be arbitrarily changed.

We note, however, that although there was an error message generated for these false positives, the system didn't crash. We doubt a true attack would cause this message to be displayed. If the attacks were crafted to display any error message (but not to crash the system), it would alert the user of a potential attack. Nevertheless, in this study, we measured these cases as false alarms. This does not invalidate the utility of this strategy; rather it demonstrates the fact that we did not have access to sufficient knowledge of the complex binary format of Word documents that inhibits the general application of the technique. We believe those with the deep knowledge of Word formats can apply this method safely to increase the security of Word documents.

5. Conclusion

In this paper, we present Arbitrary Data Transformation to detect stealthy malcode embedded in Word documents. ADT arbitrarily modifies a document's data values, and hence the byte content of maliciously embedded malcode will also be altered, if there is any. Changing normal data values might damage the display but wouldn't cause a system crash; on the other hand, system crash or errors will be observed if there is any hidden malicious executable code because its byte content is arbitrary changed and its execution flow would be damaged. Since the changed values are arbitrarily chosen and can be different

[6] Apparently this undocumented Microsoft sector was somehow discovered by the attackers who were targeting organizations sympathetic to Tibet. The reader can draw their own conclusion about how the attackers learned of this undocumented opportunity.

each time ADT is applied, this approach is secure against brute force guessing the exact transformation (e.g., which bytes are changed and by what values). The method is intended to be applied to a safe sandobox environment opening documents using an unaltered version of Word. There is no learning phase required to apply the method. However, detailed knowledge of Word document structure is necessary to identify the specific data sectors to which the method is applied.

We evaluated ADT by using 15 carefully crafted attacks provided to us by a third party; 11 caused system crashes or errors that were easy to observe, and 3 did not cause a crash but we found the "malicious behavior" (i.e. some strange files were created) was observed. For comparison, we also tested 1516 benign documents and had only 9 false positives, but using other detection methods these would not have been deemed malicious.

Since this technique requires knowledge of specific proprietary applications such as Microsoft Word, it is difficult to be generally implemented by an AV vendor without having the ability to parse the binary file content. However, if all document processing vendors performed this type of check in each of their respective applications using internal emulation or sandboxing, they would cut off an avenue of attack against their products. Hence, this technique might be included directly in MS Word as a security feature.

Acknowledgement

This material is based upon work supported in part by the US Department of Homeland Security under Grant Award Number 60NANB1D0127 with the Institute for Information Infrastructure Protection (I3P), and the Army Research Office (ARO) Under Grant Award W911NF-06-1-0151 - 49626-CI. The I3P is managed by Dartmouth College. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security, the I3P, ARO, or Dartmouth College.

Reference

1. W Arnold, G. Tesauro Automatically Generated Win32 Heuristic Virus Detection. Virus Bulletin Conference, September 2000, pp. 123-132.

2. F. Apap, A Honig, S. Hershkop. E. Eskin, S. Stolfo: Detection Malicious Software by Monitoring Anomalous Windows Registry Accesses. 3rd IEEE Conference Data Mining Workshop on Data Mining for Computer Security, 2003.

3. E. Barrantes, D. Ackley, S. Forrest: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. CCS, 2003.

4. S. Bhatkar, D. C. DuVarney, R. Sekar: Address Obfuscation: An Approach to Combat Buffer verflows, Format-String Attacks, and More. 12th USENIX Security Symposium, Washington, DC, 2003, pp.

5. V. Bontchev: Macro Virus Identification Problems. Proc. 7th Int. Virus Bull. Conf., 1997, pp. 175-196.

6. P. Busser: Memory Protection with PaX and the Stack Smashing Protector: Breaking out Peace. Linux Magazine, pp. 36-39, 2004.

7. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, H. Hinton: StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks. 7th USENIX Security Conference, San Antonio, Texas, 1998, pp. 63-78.

8. C. Cowan, S. Beattie, J. Johansen, P. Wagle: Pointguard: Protecting pointers from buffer overflow vulnerabilities. In Proceedings of the 12th USENIX Security Symposium, Washington, D.C., August 2003.

9. T. Detristan, T. Ulenspiegel, Y. Malcom, M. Underduk: Polymorphic Shellcode Engine Using Spectrum Analysis. Phrack 2003.

10. J. Evers: Zero-day attacks continue to hit Microsoft. News.com, September 2006.

11. H. Feng, O. Kolesnikov, P. Folga, W. Lee, W. Gong: Anomaly detection using call stack information. IEEE Symposium on Security and Privacy, 2003.

12. S. Forrest, S. Hofmeyr, A. Somayaji, T. Longstaff: A sense of self for unix processes. Proceedings of the 1996 IEEE Symposium on Security and Privacy.

13. X. Jiang, H. J. Wangz, Dongyan Xu, Y.-M. Wang RandSys: Thwarting Code Injection Attacks with System Service. Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems.

14. G. S. Kc, A. Keromytis, V. Prevelakis: Countering Code Injection Attacks With Instruction Set Randomization. Proceedings of the 10th ACM conference on Computer and communications security.

15. J. O. Kephart, W. C. Arnold: Automatic Extraction of Computer Virus Signatures. In Proceedings of the 4th Virus Bulletin International Conference, R. Ford, ed., Virus Bulletin Ltd., Abingdon, England, 1994, pp. 178-184.

16. D. Kierznowski: Backdooring PDF Files. September 2006.

17. H.-A. Kim, B. Karp.: Autograph: toward automated, distributed worm signature detection. In Proceedings of the 13th USENIX Security Symposium, August 2004.

18. O. Kolesnikov, W. Lee: Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. USENIX Security Symposium. 2006, Georgia Tech: Vancouver, BC, Canada.

19. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, G. Vigna: Automating mimicry attacks using static binary analysis. Proceedings of the 14th conference on USENIX Security Symposium.

20. J. Leyden: Trojan exploits unpatched Word vulnerability. The Register, May 2006.

21. K. Lhee, S. J. Chapin: Type-Assisted Dynamic Buffer Overflow Detection. 11th USENIX Security Symposium, 2002, pp. 81 - 88.

22. W. J. Li, S. J. Stolfo, A. Stavrou, E. Androulaki, A. Keromytis: A Study of Malcode-Bearing Documents. DIMVA, 2007.

23. W.J. Li, S. J. Stolfo SPARSE: A Hybrid System to Detect Malcode-Bearing Documents CU Tech. Report, Jan 2008.

24. J. Newsome, B. Karp, D. Song: Paragraph: Thwarting signature learning by training maliciously. In Proceedings of the IEEE 9th International Symposium on recent Advances in Intrusion Detection (RAID), 2006.

25. C. Parampalli, R. Sekar, R. Johnson: A Practical Mimicry Attack Against Powerful System-Call Monitors. ACM Symposium on Information, Computer and Communications Security 2008.

26. R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading Worm Signature Generators Using Deliberate Noise Injection. In Proceedings of the IEEE Symposium on Security and Privacy, 2006.

27. H. Shacham, M. Page, B. Pfaff, E.J. Goh, N. Modadugu, D. Boneh: On the Effectiveness of Address-Space Randomization. Proceedings of the 11th ACM conference on Computer and communications security,pp 298--307, 2004.

28. S. Singh, C. Estan, G. Varghese, S. Savage: Automated Worm Fingerpriting OSDI 2004.

29. Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, S. J. Stolfo: On the Infeasibility of Modeling Polymorphic Shellcode for Signature Detection. Proceedings of the 14th ACM conference on Computer and communications security 2007.

30. S. J. Stolfo, W. J. Li, K. Wang: Towards Stealthy Malware Detection. Malware Detection Book, Springer Verlag, (Jha, Christodorescu, Wang, Eds.), 2006.

31. P. Szor The Art of Computer Virus Research and Defense. Symantec Press, 2005.

32. D. Wagner, P. Soto: Mimicry attacks on host-based intrusion detection systems. Proceedings of the 9th ACM conference on Computer and communications security, CCS 2002.

33. K. Wang, J. Parekh, S. J. Stolfo: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. Proc. Int. Conf. on Recent Advanced in Intrusion Detection, RAID06, Sept 2006.

34. Attackers hose down Microsoft's Jet DB Engine. http://www.channelregister.co.uk/2008/03/26/jet_database_engine_security_flaws

35. CVE-2006-3647

36. CVE-2007-3899

37. CVE-2006-5994

38. Cyber Attacks Target Pro-Tibet Groups. http://www.washingtonpost.com/wp-dyn/content/article/2008/03/21/AR2008032102605.html

39. K2. ADMmutate. 2001 Available from: http://www.ktwo.ca/security.html

40. Microsoft Office Reference Schemas http://rep.oio.dk/Microsoft.com/officeschemas/SchemasIntros.htm

41. Microsoft Office Word 2007 Binary File Format Documentation Microsoft 2007.

42. Microsoft Security Bulletin. http://www.microsoft.com/technet/security/advisory/912840.mspx

43. Microsoft Word Document Parsing Buffer Overflow Vulnerabilities http://secunia.com/advisories/12758/

44. Steganalysis http://niels.xtdnet.nl/stego/

45. P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS), pages 59–68, 2006.

46. A. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. In Proceedings of the USENIX Security Conference, 2006.

47. K. Natvig: SandboxII: Internet Norman SandBox Whitepaper, 2002

48. C. Willems, F. Freiling, T. Holz: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security and Privacy Magazine, April 2007, Vol. 5, No. 2, pp. 32-39