CryptoBlock: End-host Cryptographic Malware Detection *and* Mitigation System

Eugene Kolodenker, Brett Kaplan, Alexandrea Mellen, Kanimozhi Murugan, Raghavendra Punichittaya
Boston University
Boston, USA
{eugenek,bkaplan,almell,mkani,raghavvp}@bu.edu

ABSTRACT

Malware is an ever increasing threat to the Internet and its users. Cryptographic malware, also known as cryptomalware, cryptoviruses, and ransomware is a rising form of malware. Ransomware uses cryptography to encrypt victim's files and extort money from its victims in exchange for decryption. In the past ransomware has been juvenile and poorly designed; however, new ransomware strains are proving to be significantly more adaptable and expansive. By detecting cryptography happening on the victim's computer machine, we are able to substantially mitigate damage, and in some cases, prevent it entirely.

I. Introduction

Cryptographic malware also known as cryptomalware, cryptoviruses, and ransomware is an increasing threat for Internet users. Research into cryptographic malware is limited, though initial documentation begins as early as 1996 [1]. Since then, extortion-based attacks (ransomware) and other forms of cryptomalware have risen dramatically in popularity. Thus, ransomware has become significantly more sophisticated through the use of better encryption systems and an increase in targeted files and the advent of targeting network files [4]. Popular variants of ransomware, CryptoLocker and CryptoWall, are estimated to have collectively extorted over a million dollars from their victims [2].

We present a system, *CryptoBlock*, designed to detect and mitigate cryptographic malware. *CryptoBlock* is a system for live, continuous detection and mitigation against cryptographic malware on a Windows 7 32-bit end-host. Our system is designed for completely autonomous detection of cryptographic malware.

Traditional detection-based passive and active protection from antivirus programs does not attempt to mitigate the damage caused by cryptographic malware. Typically, these methods do not allow the user to restore files encrypted by malware to their unencrypted state. Additionally, current forms of passive detection do not specifically target unusual cryptographic behavior, and instead rely on signatures for their detection. Other works have focused on active malware scanning of files with a focus on security analysis [3], [5], [6]. Our implementation takes a different approach to provide continuous protection and mitigation.

Our work builds upon the research of Matenaar et al. [3] and consisting of four modules: API hooking, constant search, entropy difference, and a novel ASCII string detection. A key difference in [3] and *CryptoBlock* is that instead of targeting security analysts and making use of an emulated environment, we design our system for a live end-host Windows 7 system. We focus our priorities on detecting and logging cryptographic malware live and passively on the end-host to help mitigate the damage caused.

II. System Details

CryptoBlock consists of four major modules. We discuss Cryptographic API hooking done via Microsoft Detours, cryptographic constant detection, entropy detection for encrypted data, and the existence of human readable strings in data files.

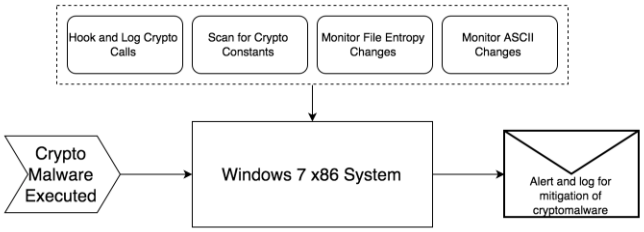

Fig. 1. CryptoBlock overall system design.

A. API Hooking

A broad range of cryptographic functions are available as part of today's operating systems [3]. Microsoft Windows 7 comes with a fully capable cryptographic library, CryptoAPI [8]. The library is documented, with sample code, and a long history of usage [1] - as such it proves to be an easily integrated and popularly used library in cryptomalware. With our focus on the Windows 7 end-user we utilized Microsoft Research's Detours [16] to implement API hooking (function rerouting) into the Windows CryptoAPI.

API hooking is a technique that is used to alter the behavior of an operating system, of applications, or other software components by intercepting function calls and overwriting them with the "hook". Microsoft's Detours allowed us to hook directly into the Windows API. Furthermore we utilized DLL injection to have our hooks launch at the run time start of most

applications on Windows 7. Through the use of the registry key, AppInit_DLLs we load every process that dynamically links to User32.dll with our CryptoHook.dll. A limitation of this approach is that while most applications on Windows hook to User32.dll, not all do. As such if the malware manages to avoid hooking into User32.dll by limiting their use of the Windows API to very select few functions our injection does not work.

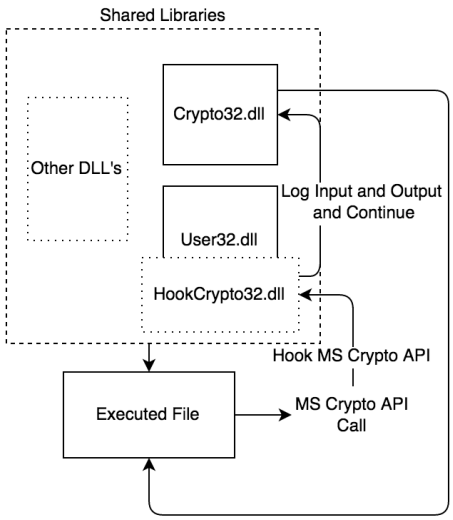

Fig. 2. API hooking and DLL injection on the Windows 7 platform with *CryptoBlock*.

We analyzed 128 samples of cryptographic malware for details on commonly used API calls. PEStudio [8], and Cuckoo Sandbox [7] were utilized to perform static and dynamic analysis respectively of each sample. Static analysis proved useful on binaries that were not obfuscated with methods such as symbol stripping or packing. However, obfuscation concealed significantly less against our dynamic analysis environment utilizing Cuckoo Sandbox. With these tools we catalogued a list of the commonly used Windows CryptoAPI functions, and implemented our own functions to hook into the most likely reversible and useful functions. By consulting Microsoft Developer documentation [9] we were able to understand which functions had potential to be reversed (i.e. symmetric encryption) and mitigated against. *CryptoBlock*'s hooking module logs any of the hooked function calls to the CryptoAPI and their inputs and outputs. A walkthrough of an encryption reversal through the use of logged function calls, and their inputs and outputs can be seen in Fig. 3.

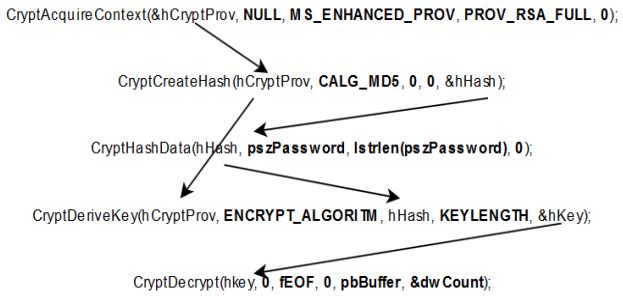

Fig. 3. Decryption done through logging of previous crypto function calls, inputs, and outputs.

B. *Constant Scanning*

Cryptographic constants can either be statically stored in the executable, often times the .data section in the Windows PE format, or dynamically generated on the fly and stored into memory during runtime. We focused research to the common crypto frameworks and libraries, including miracl, crypto++, openSSL, and cryptoAPI.

Table.1 Statistic information of typical Hash functions and Symmetric algorithms

Cryptography algorithms	Percentage of bitwise and arithmetic instructions	Static basic blocks	Loops
Md5	99.83%	1	No
Md5mac	99.74%	2	No
Md4	99.81%	1	No
Sha1	99.78%	1	No
Haval	96.93%	1	No
Gost	98.87%	1	No
Ripemd	99.65%	1	No
Des	96.03%	4	Yes
Blowfish	87.12%	3	Yes
Cast	98.93%	4	Yes
AES	94.73%	3	Yes
RC6	84.64%	4	Yes

Fig. 4. Typical cryptographic functions and the percentage of arithmetic in their basic blocks [10].

Additionally, cryptographic algorithms often apply fixed constants, such as initialized values of hashed functions, and sometimes use them as an assigning value function for the main encryption function. Other research [20] indicates the possibility that certain implementations of crypto algorithms using fixed constants might use different values of these constants (without affecting their security) in order to avoid detection. Considering this, we resorted to analyzing the binary statically to achieve better results. The research helped to obtain a more reliable source of crypto constants signatures, which were used in our work.

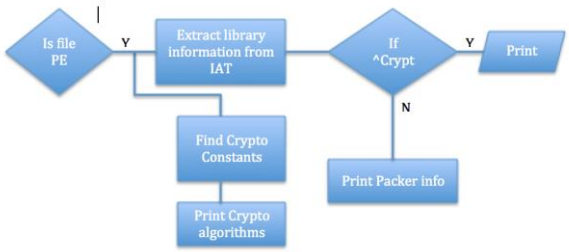

Fig. 5. Workflow for the constant scanning module.

The python program watches the user directories recursively for created files. The full file path of any file created under a specific directory is stored and several analyses are performed, including:

Executable check:

The executable check identifies whether the first two letters of the file contain the characters "MZ" from the MZ headers

(from the legendary Microsoft Architect - Mark Zbikowski) [11].

Import Address Table(IAT):

The IAT is a table of function pointers filled in by the windows loader as the DLLs are loaded. By using the *'pefile'* python module, the program checks the `.rdata` section of the IAT for functions that begin with "Crypt".

Packed file:

Using the *'peutils'* python module and the signatures from userdb.txt, the subject file is scanned for packers. The search uses a text file named '*userdb.txt'* [12] This is a signature file used to support the peutils packer detection which contains 1832 signatures that identify most common packers, cryptors and compilers. There are multiple similar files available on the internet and one such is used by PEiD.

Encryption Search:

Signsrch [13] is a program to check the PE for crypto constants, crypto functions, codecs, and to indicate more information about the file. The program checks for possible crypto constants used in the binary, which can then be used by disassemblers to follow cross-references and identify the key(s) used.

In order to invoke this, one should create a copy of the file under the program's observed directories (*'C:\Users\'* in our case). The output is in a command prompt which comprises of the PE, the crypto functions used from the MS Crypto API library, packed details, and the crypto constants.

C. Entropy

Entropy analysis is a highly effective method of cryptomalware detection in real-time [19]. Entropy is the measure of randomness or unpredictability in a sequence. When files are encrypted, the contents increase in entropy values over a threshold expected for unencrypted normal text and PDF documents. Entropy calculation of file contents can be calculated in a variety ways such as Shannon's entropy on the frequency of occurrences of each ASCII character, or binary entropy of the file contents through recognition of repetitive patterns in the binary sequence.

Binary entropy is calculated when the contents of the file are converted to a binary sequence. Entropy is calculated using the normalized difference between the expected repetition of data in a normal file to the actual observed repetition in the file under analysis. Implementation of binary entropy analysis was was done using a statistical test suite used to compute the discrete fourier transform (DFT) of the binary sequence through computation of the peak threshold (T), expected out of the sequence, and the actual peak in the sequence. T - determined as the 95% peak threshold values. Under the assumption of randomness, 95% of the peaks in the sequence should have height less than T. If the entropy value (d) from these figures are too low, then it means there were very few peaks less than T (<< 95%) and more peaks with values greater than T(>>5%). The threshold value varies with type and size of the file. Thus to draw a range as to detect encrypted files, complementary error function (erfc) for the calculated entropy value is computed. Considering the low 'd' value as a positive reference for random sequence, erfc value would range >= 0.01 for a positive randomness test.

Our current implementation of entropy analysis is limited by false positives for extremely small files. A file with the contents "SSN: 0123456789″ would be detected as encrypted due to extremely high entropy, because within the 14 characters, only S is non-unique. Additionally, the current implementation offers no distinction between legitimate user encryption and unauthorized encryption.

D. ASCII Analysis

Regular text documents consist largely of characters in the ASCII character set, whereas encrypted files consist of random data with very little pattern. We originally designed a module to compare the amount of ASCII characters in a file to the amount of total data; if a threshold was not reached we determined the file was encrypted. However, this method of detection proved to be fairly ineffective as more often than not, encrypted files went up in ASCII value due to the randomness and ASCII taking up a large chunk of the 0x00 to 0xFF byte space.

We instead changed our design to detect proper strings through use of GNU *strings* [13]. We scanned written files for the presence of strings greater than four characters and created a threshold based on the size of the file and the number of strings in the file. We set up event driven file write monitoring through the use of *inotify-win* [12]. Upon file medication, a callback function is called that would trigger our ASCII analysis.

III. Results, Challenges, and the Future

The development of *CryptoBlock* was not without challenges and design limitations. We designed for real cryptomalware, but due to time constraints our testing was limited to one malware sample and two simpler encrypting programs. CryptoBlock was able to detect and log cryptographic function calls, files changed, and constants discovered for all three of test cases: encrypting a text file with a password, encrypting a text file with a randomly generated key, and executing our sample CryptoLocker binary. As such we are convinced that with time we could further test our system and determine that it is effective in detecting and potentially mitigating cryptomalware.

Our current API hooking implementation is limited to the Microsoft CryptoAPI. A major improvement would be to first expand into the other potentially cryptographic API's installed with Windows 7 such as Secur32. Afterwards expanding into hooking into statically linked library functions would greatly increase the effectiveness of the API hooking approach to not only malware that is dynamically linked to Windows APIs. Furthermore a more advanced DLL hooking system leveraging perhaps CreateRemoteThread, or SetWindowsHookEx [15] increase the many downfalls of AppInit_DLL hooking [16]. Additionally, we ran into several unexpected issues after API hooking was launched. These issues ranged from the Visual Studio compiler being unable to work, web downloaders no

longer functioning, and the inability to reboot our systems. Because of these issues we'd have to do additional research into the effects of API hooking to truly deliver a bug free tool. We suspect that many programs do a CRC check of the functions, or a time analysis, or another form of mitigation against cryptographic attacks.

Our cryptographic constant scanning module would be greatly improved through the use of API hooking to perform real time scanning for signatures of crypto constants.

We are also interested in adding additional heuristics to our current entropy analysis module to reduce the amount of false positive in small files. One such heuristic is limiting the entropy analysis to calculating the amount of "regular" unicode characters of the Windows user's default character set (i.e. English, Hungarian, Russian, etc.). Additionally we'd like to make this module more efficient by hooking into the Windows API and implementing an event driven design to detect file writes. Furthermore our ASCII strings detection module would be improved through further research into determining a precise threshold for determining encrypted files.

IV. Conclusion

CryptoBlock provides a broad range of modules for passively detecting and mitigating cryptographic malware on a modern Windows 7 end-host. Previous work in dynamically detecting cryptographic malware has focused on emulated environments with the security analyst in mind. We believe through the use of Windows API hooking, cryptographic constant scanning, file entropy analysis, and file ASCII analysis we are able to provide an effective novel take on an anti-cryptomalware system with the common end user in mind.

Acknowledgment

We would like to thank Professor Manuel Egele for providing us with the cryptographic malware samples.

References

[1] Adam Young, Moti Yung. Cryptovirology: Extortion-Based Security Threats and Countermeasures. In IEEE Symposium on Security and Privacy (1996)

[2] Vadim Kotov, Mantej Singh Rajpal. Understanding Crypto-Ransomware. In Bromium (2014).

[3] Felix Matenaar, Andre Wichmann, Felix Leder, Elmar Gerhards-Padilla. CIS: The Crypto Intelligence System for Automatic Detection and Localization of Cryptographic Functions in Current Malware. In IEEE Malicious and Unwanted Software (2012)

[4] Lloyd Bridges. The changing face of malware. In Network Security (2008).

[5] Ulrich Bayer, Andreas Moser, Christopher Kruegel, Engin Kirda. Dynamic analysis of malicious code. In Journal on Computer Virology (2006).

[6] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and Efficent Malware Detection at the End Host. In Proceedings of USENIX Security Symposium (2009)

[7] Cuckoo Sandbox: Automated Malware Analysis. http://www.cuckoosandbox.org/

[8] Pestudio. http://winitor.com/

[9] The Cryptograph API, or How to Keep a Secret. https://msdn.microsoft.com/en-us/library/ms867086.aspx

[10] Li Ji-Zhong, Yin Qing, Jiang Lie-Hui, Jia Xin-Hai. Analysis of Cryptographic Algorithms' Characters in Binary File. In International Conference on Parallel and Distributed Computing, Applications and Technologies (2012)

[11] Peering inside the PE. https://msdn.microsoft.com/en-us/library/ms809762.aspx

[12] PEiD Signatures. https://code.google.com/p/reverse-engineering-scripts/downloads/detail?name=UserDB.TXT

[13] Signsrch. http://aluigi.altervista.org/mytoolz.htm

[14] Inotify-win. https://github.com/thekid/inotify-win

[15] GNU Binutils. http://www.gnu.org/software/binutils/

[16] Microsoft Research. Detours. http://research.microsoft.com/en-us/projects/detours

[17] Infosec Institute. API Hooking and DLL Injection on Windows. http://resources.infosecinstitute.com/api-hooking-and-dll-injection-on-windows/

[18] Microsoft Dev Center. AppInit DLLs and Secure Boot. https://msdn.microsoft.com/en-us/library/windows/desktop/dn280412%28v=vs.85%29.aspx

[19] Matthew M. Shannon. Forensic Relative Strength Scoring: ASCII and Entropy Scoring. In International Journal of Digital Evidence (2004)

[20] Joan Calvet, Jose Fernandez, Jean-Yves Marion. Cryptographic function identification in obfuscated binary programs (2012)

Appendix

A. Roles

Eugene Kolodenker organized the team, and designed and led the development of *CryptoBlock*. Additionally he was the primary editor and writer for the paper. He contributed the API hooking module.

Brett Kaplan researched and statically analyzed different malware samples to better understand how ransomware works. Using this data he provided the team with the information necessary to make each module more robust. He participated in testing HookCrypt. He also helped Raghav in doing research regarding locating the magic constants for cryptographic algorithms, scanning a process' memory leading to file scanning, and looking for other string identifiers.

Alexandrea Mellen developed her own ransomware to understand exactly how to identify ransomware on a deeper level. She also implemented and tested the ASCII strings analysis module, including understanding the failings of the original method and pivoting to a new approach. Additionally she helped with the editing of the final report.

Kanimozhi Murugan worked on the file entropy detection method. She was responsible for the research and development of the module to provide effective encrypted file detection. She brainstormed on the packed malware detection by calculating Shannon's entropy and researched other existing detection methods using binary entropy.

Raghavendra Puninchittaya worked on the cryptographic constant scanning. He obtained crypto constants used from the memory when a program executes. All the functionality in his module helps to understand the ransomware to perform further analysis and supports the key recovery.

B. *Source Code*

Our source code can be found at https://github.com/EC700/Charlie-2. We implemented a modular design with each module able to be isolated and ran separately from the others. Referring to each module's README is sufficient for installation, running, and analyzing.