2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress

Active Malware Countermeasure Approach for Mission Critical Systems

Zachary Thomas
Mississippi State University
Starkville, MS 39762
Email: zbt5@msstate.edu

Sherif Abdelwahed
Virginia Commonwealth University
Richmond, VA 23284
Email: sabdelwahed@vcu.edu

Abstract—**This paper presents a cyber-defense system, named the Active Malware Countermeasure system, aimed at maintaining the operation of mission-critical systems by mitigating damage after compromise has occurred. This system is designed to implement techniques that will nullify the effect of malicious actions taken against mission-critical systems. Diverting identified malicious functionality away from the mission-critical system to sacrificial virtual machine(s) (VM) effectively nullifies malware. The proposed system utilizes hooking techniques to intercept malicious operations before they execute in the critical environment. The hooks are then used to facilitate diversion of malicious functions to a VM. Virtualization is used to create a safe, isolated environment where the malicious functionality can be executed without affecting the critical environment. Return values and information associated with the function execution can then be sent back to the malicious process, so its execution can proceed as normal. The maintained execution of malware in this hooked state allows the mission-critical system to continue operating as damaging functionality is diverted to the safe environment, minimizing down-time of critical operations despite compromise.**

1. Introduction

Cyber-attack defense is a heavily-researched and reviewed area in the current era, the Digital Age. The public and private sectors' reliance on computers, particularly in their usage of digital storage, has driven the focus to securing these computers from compromise. Compromise in the form of malware infection often leads to data theft, manipulation, and/or destruction. Additionally, compromise can negatively affect critical operations being performed by infected computers. Countermeasures developed by the research community to protect data and operations can generally be categorized into two approaches, securing targeted systems against attacks, and responding gracefully to attacks launched against targeted systems [1]. The majority of research design and implementation of cyber-defense systems follows the former model. Modern research has been unable to develop a perfect solution employing this method. Due to shortcomings inherent to this model, systems are compromised and must be restarted or reinstalled to fully remove any and all malware. Systems carrying out crucial operations, called mission-critical systems here, are often a target of these attacks and also must be taken offline, but the resulting, associated impact from halting crucial operations is greater.

To take the latter approach at cyber-defense with the primary objective of enabling a mission-critical system to continue running after infection, this paper proposes the Active Malware Countermeasure (AMC) system. The AMC system is designed to identify and negate components of a running process capable of harming a mission-critical system. Functionality that has been determined to be potentially malicious is diverted from the mission-critical system to a safe environment, a sacrificial virtual machine (VM), where the functionality can be executed without impacting critical operations. After the diverted functionality has executed in the sacrificial VM, any relevant values created or obtained as a result of its execution are returned to the malicious process running in the mission-critical environment. The malware then proceeds with its execution until other specified functionality is encountered, in which case it is also diverted, or until it terminates. The interception of process execution to divert potentially malicious functionality allows malware to persist with a reduced negative impact on the mission-critical system, significantly reducing or eliminating the urgency to take the mission-critical system offline to remove the malware.

There are a few notable aspects of this system's approach at the diversion process. First, virtualization is used, in the form of a sacrificial VM, to receive the diverted functionality and execute it. A VM in this capacity is beneficial first and foremost because it is encapsulated in an environment isolated from critical operations. Malicious actions can be performed against the VM without harming the underlying system. Additionally, after any damage has been done to the VM, it can be restored to a clean state through the use of snapshots. Next, the implementation of diversion in a running malicious process begs the following question: "if control of execution within the process's context is required to perform diversion, why not just terminate the process or run an infinite sleep loop rather than diverting identified functionality, allowing other functionality to be executed?" While these alternatives would stop the process from executing malicious sections of code, they would also disrupt its execution. Suppose after dropping and starting malware execution, a malware dropper simply waited on a message from the main process, and if it did not receive it after a specified amount of time, it restarted the main process. In this way, the malicious process would be restarted and may fail to be detected on this execution. Many variations of this idea can be constructed that evidence the utility of diversion rather than termination. Allowing the malicious process to continue to execute but intercepting and diverting potentially malicious sections of code also allows the mission-critical system to continue operating with a reduced negative impact.

In this paper, "hooking" techniques are utilized to perform the diversion of identified target code sections. For the purposes of this project, "hooking" is described as modifying the path of execution of a running process to intercept function calls [1]. Several techniques exist for accomplishing hooking in this way. When an interrupt is triggered in running code, an OS-level table is consulted to determine the code to jump to in handling the exception. When a module is loaded into memory, a table of imported functions is written to with the address of each function to jump to when a function call is encountered. These tables can be overwritten with the address of hooking code in memory, so this code will be jumped to rather than the OS or library code, effectively hooking the running process. If these tables are not writable, portions of the application code can often be overwritten directly to jump to hooking code. Using one or a combination of these methods, the AMC system can divert identified functionality from the mission-critical system to a sacrificial VM.

This paper is organized as follows. Section 2 introduces related works, including those that take the detection and prevention cyber-defense approach, and compares and contrasts those works to this one. Section 3 focuses on the design of the AMC system. It begins with a description of relevant aspects of the Windows operating system and then details how these components can be utilized to achieve our desired functionality. Section 4 describes how the AMC design was implemented in software and some challenges encountered during the coding process. Section 5 discusses the importance of testing in the appropriate environment, describes initial testing that has been performed, and presents future work to improve and expand the AMC system's capabilities.

2. Related Work

As mentioned previously, cyber-attack defense continues to be a heavily-researched area. As a result, several works have been created with similar missions to our project. The majority of these works focus on malware detection and prevention, the first approach to cyber-defense, in the form of intrusion detection and anti-virus systems [2] [3] [4] [5]. To describe a recent work employing this strategy, an intrusion detection system for connected vehicles was proposed [6]. This system uses a combination of a cloud-based and on-board IDS to detect the presence of malware remotely and locally before its execution. These strategies differ from our system in their foundational approach to defense against malware. Though much research in the past has focused on this first approach, some recent projects have embraced the second approach to cyber-defense, responding to malware infection after compromise has occurred.

A self-defense system reminiscient of that existing in the human body was proposed to protect against malware as well as DoS attacks [7]. This system differs from our system in that it first detects the presence of an attack, but following detection, it attempts to eliminate malware from the system rather than nullifying its exeuction. A cyber-defense system utilizing a blacklist to block internet connections made by malware was proposed and evaluated [8]. Blocking of malware network traffic prevents communication with a command and control server, effectively neutralizing malware dependent on commands. Though this net effect achieves a goal similar to that of our project, it differs in that our project is designed to also neutralize malware not dependent on external commands. In addition to malware defense, other work related to this project uses virtualization to enhance security [9] [10]. The isolation of VMs from the main operating environment provides a safe path for malware execution and is essential to the functioning of our cyber-defense system.

3. AMC System Design

This section will describe in details the design of the AMC system's hooking technique. Background on the infrastructure and concepts that make this design possible will also be included. Before describing the implementation details of the AMC system, a few assumptions should be highlighted that were considered when developing the system. First, the mission-critical system running the AMC system runs a version of Windows NT operating system. This assumption restricts the scope of potentially malicious functionality to the Windows API as malware is likely to use the target OS API to perform harmful actions. Additionally, the design of the AMC system assumes the target malware has already infected the mission-critical system, it is executing, and its main process of execution has been detected and identified. Malware detection is recognized in this work as a separate, heavily-researched problem. Lastly, this initial version of the project assumes the sacrificial VM has been created and configured to be identical in OS to the mission-critical system.

3.1. Windows API

Because the scope of OS API to consider has been restricted to the Windows API, a description of the types of functions believed to be used maliciously and examples of those will be included in this section. The Windows API is utilized by both legitimate and malicious applications to interact with the OS. Each system running a particular version of Windows contains the same API, a useful fact for designing the diversion process that will be described in more detail in a future section. Sikorski and Honig [11] provide insight on common capabilities included in malware and the API components often utilized to carry out malicious functionality. First, malware often manipulates the Windows file system to create, delete, read, or write files. Examples of Windows API functions that perform these actions include `CreateFile`, `DeleteFile`, `ReadFile`, and `WriteFile`. Deleting important files is obviously undesirable, but other examples of undesirable actions made possible with these functions include dropping more malware and reading sensitive information. Next, many malware families require communication over a network with a command and control (C2) server to receive commands and send responses. There are many Windows API functions comprising several libraries capable of networking operations, but a few to note include `connect`, `send`, and `recv` in the Winsock library and `InternetOpenUrl`, `InternetReadFile`, and `HttpSendRequest` in the WinInet library. Lastly, malware often creates processes and runs shell commands to make changes to an infected system. The `CreateProcess` and `ShellExecute` functions are just two of many such functions capable of causing new processes to execute. These malware capabilities can all be used to cause harm to a mission-critical system; therefore, they comprise the initial list of functionalities to be hooked in the diversion process.

3.2. Portable Executable Binary File Format

The Portable Executable (PE) binary file format is the standard format of all binaries, or executables, that run on Windows NT systems. PE files are broken into headers, mostly comprised of the executable's technical details, and sections, comprised of the executable's contents [12]. Of the sections in a PE file, the Imports section is most important for the purposes of hooking as it contains the table of imported functions and their entry points. Figure 1 shows a high-level overview of a PE file [13].

The location of the Imports section in relation to the other sections can be seen in the figure. The Imports section is comprised of a series of import descriptors, where each descriptor represents a particular Dynamically-linked Library (DLL) linked by the executable. In each descriptor, an Import Name Table (INT) and Import Address Table (IAT) exist which contain information about the imported functions in each DLL. These elements of import descriptors will be described in more detail in the next section.

3.3. IAT Hooking

Figure 2 shows the components of an import descriptor before the module is loaded into memory.

The INT is labeled here as the *HintName Array*. The INT and IAT both point to the same array of structures, containing an ordinal, function name pair, before the module is loaded into memory. When a Windows executable loads into memory, the addresses of imported functions are found in the corresponding DLL's export address table (EAT) and filled into the application's IAT [14]. When an API function

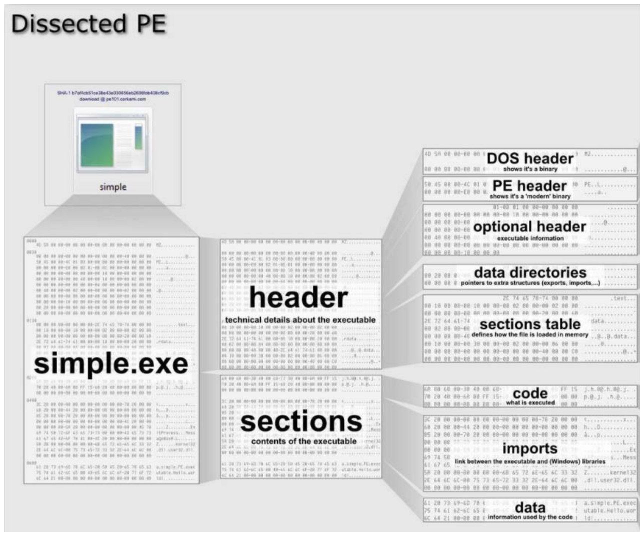
Dissected PE
simple
simple.exe
header
technical details about the executable
sections
contents of the executable
DOS header
shows it's a binary
PE header
shows it's a 'modern' binary
optional header
executable information
data directories
pointers to extra structures (exports, imports...)
sections table
defines how the file is loaded in memory
code
what is executed
imports
link between the executable and (Windows) libraries
data
information used by the code

Figure 1. PE file layout [13]

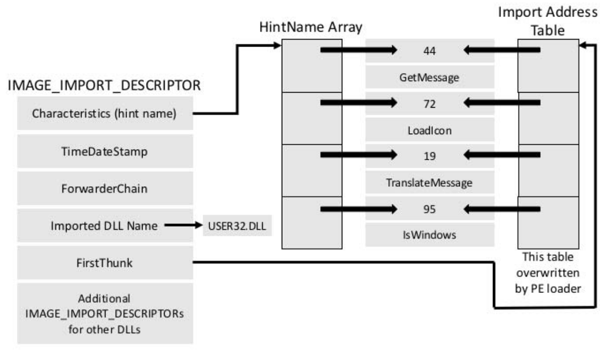

Figure 2. Image import descriptor data members [12]

call arises within an executable, the function's address is looked up in the IAT and loaded into the instruction pointer to jump to the function entry point. This look-up in the IAT for a function address occurs each time an API call is made, so the loader does not have to patch the address of the imported function into the code at each call [12]. Consequently, overwriting the address written into the IAT with the address of a different function in memory causes execution to jump to this alternate function. This procedure is known as an IAT hook, one of the hooking techniques mentioned previously. IAT hooking is often used by malware authors to intercept and manipulate the operations of legitimate applications. This same concept can be used against malware to divert its malicious functionality from a mission-critical system to a sacrificial VM. Figure 3 shows the proposed workflow of the AMC system, providing high availability to the mission-critical system as a result of the malware's application code being diverted to a sacrificial VM using IAT hooking.

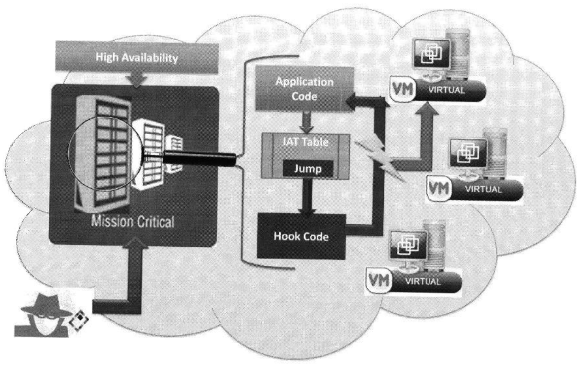

Figure 3. AMC workflow defending mission-critical system after compromise [1]

4. AMC Implementation

This section will describe how the AMC system and its diversion capabilities were implemented.

4.1. DLL Injection

In order to successfully IAT hook a running process, code must be executed within the process's context to overwrite the function addresses in the module's IAT. One method of achieving this code execution is to inject a DLL into the process's address space [15]. When a DLL is initially loaded into a process's address space, it is given the opportunity to execute code within its `dllmain` function. This control of execution can be utilized to overwrite imported functions' addresses. The DLL injection is coordinated through the use of a separate, standalone process. Figure 4 shows a flowchart of this injection process.

In the figure, *program.exe* is the standalone process, *inject.dll* is the DLL to be injected, and *victim.exe process* is the running malicious process. First, *program.exe* makes the `OpenProcess` call to gain a handle to *victim.exe process*. This handle must be gained with privileged access to the process, so that memory can be written and a remote thread can be created in its address space; in the implementation, `PROCESS_ALL_ACCESS` was used to gain all access rights. Then `GetProcAddress` is called to acquire the address of the `LoadLibrary` API call for future use. Next `VirtualAllocEx` is called to allocate memory within the process's address space. `WriteProcessMemory` is used to write the name of the DLL, *inject.dll*, into the memory region allocated. Lastly, a remote thread is created in the process's address space with `CreateRemoteThread`. This thread is passed arguments to make the `LoadLibrary` API call on *inject.dll* to successfully load the DLL into *victim.exe process*'s address space. Using this flow of sequential API calls, our DLL is injected into the malicious process where it can implement the IAT hooks.

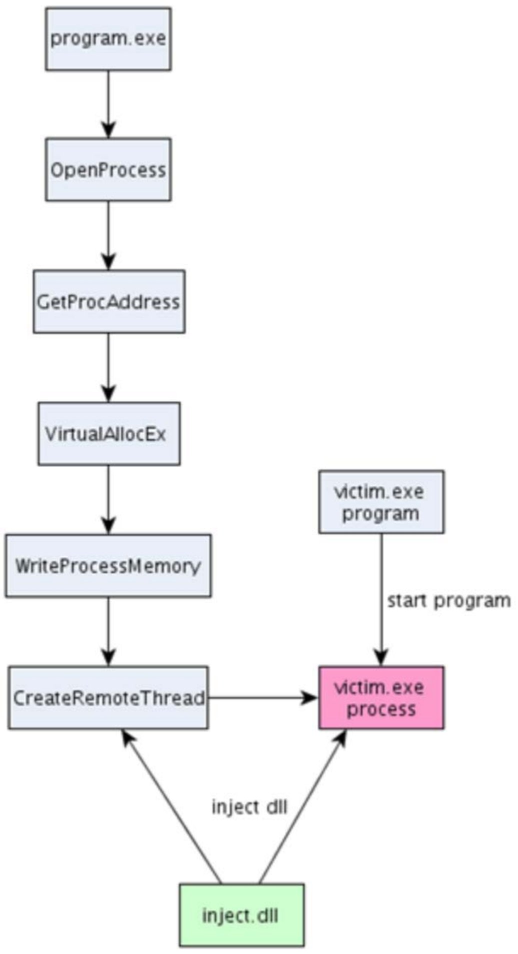

Figure 4. DLL injection flowchart [15]

4.2. IAT Hook Code Description

Now that control of execution has been obtained in the malicious process's address space using the injected DLL, the code can be used to overwrite the imported functions' addresses, effectively implementing the IAT hook. This operation will involve traversing a PE file; Figure 1 can be referenced to follow the traversal description. The first step in this IAT hooking process is to load a handle to the module, or the malware executable [14]. The optional header should then be accessed as an offset from the start of the module, or the DOS header. The address of each component will be calculated in this way, as an offset of this image base. The optional header contains the *DataDirectory* array which can be used to retrieve the Relative Virtual Address (RVA) of the Imports section and, resultantly, the start of the Imports section. The Imports section is comprised of an array of import descriptors, referenced in Figure 2. In functions that are exported by name, the INT is then used to traverse through the imported functions contained in the current DLL. Once a targeted function is found, the address of its entry point is obtained from the IAT and overwritten.

If a function is exported by ordinal rather than by name, the *Name* field of the INT is not written, so it must be traversed differently. The IMAGE_ORDINAL_FLAG constant can be used to determine if a function is exported by ordinal [16]. When a function is encountered that satisfies this condition, the *Hint* field of the INT should be read to obtain the ordinal value, and it should be used to identify the function rather than the function name. The address can be overwritten the same way as functions exported by name. This traversal and overwriting should be continued for all imported functions in all import descriptors to effectively hook all targeted potentially malicious functions. After the addresses have been overwritten, whenever a targeted function call arises during malware execution, the flow of execution will jump to the new address written to the IAT, and the corresponding hook function will run in place of the API function.

4.3. API Hook Functions

Once the IAT hooks are in place and our functions are called instead of the original API functions, the actual diversion from the mission-critical system to the sacrificial VM can be performed. For clarification, the functionality described in this section occurs each time a targeted API function call is encountered during the malicious process's execution. Because the scope of targeted functions capable of causing harm has been restricted to the Windows API, excluding user-defined functions, no executable code has to be copied from the mission-critical system to the sacrificial VM. The API is already present on the VM with the same OS. The only elements that are necessary to send to the sacrificial VM are the name of the function being hooked and the parameter values passed to the function. Using these components, the sacrificial VM can recreate the original Windows API call and execute it. Tailored code must be developed for each hooked API call because each function has its own associated arguments. A description of the diversion code follows.

Whenever an identified API function call is encountered in the malicious process, the hooking code creates a TCP connection from this context to a listening server on the sacrificial VM. Parameter values are cast to a transferable format. Furthermore, any structures storing essential information must be deconstructed into their data members to obtain the individual, important values. Determining which parameter values and data members are necessary to recreate a particular Windows API call without errors required research about each hooked function [17]. After the function name and essential, associated argument values are obtained, all of these values must be transmitted to the sacrificial VM.

Once the server has accepted a TCP connection from the malicious process client, it begins receiving information about the current hooking function. It first receives the name of the function being hooked and then the function argument values. These argument values must be parsed, cast back to their respective types, and used to reconstruct any structures that were deconstructed client-side [17]. The Windows API call can then be made with the same necessary argument values passed in the malicious process. After the function call has been executed and returned, any useful resources that were allocated and returned as a result can be stored for future use. These resource values and/or the function return value can then be sent back to the malicious process client over the same established TCP connection.

4.3.1. Unique Resources Challenge. One DLL containing numerous functions of interest to the project for networking capabilities is the WinInet library. Many WinInet library functions create a resource in the form of an internet handle, HINTERNET, as a result of their execution. This internet handle is to be taken in as an argument to proceeding function calls associated with this connection. This dependence of some function calls on the internet handle from other function calls results in a hierarchy of calls to accomplish certain tasks. Figure 5 depicts an example of a hierarchy of WinInet function calls.

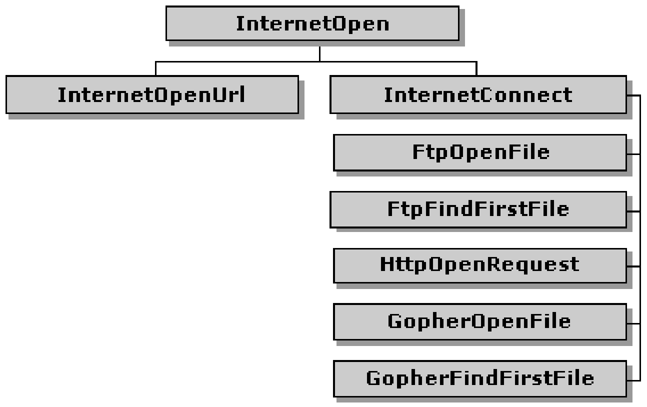

Figure 5. WinInet HINTERNET Hierarchical Structure [18]

The relevance of this hierarchy to the project is that each dependent function relies on a unique HINTERNET handle. These handles need to be stored on the sacrificial VM server, but there must be a way to uniquely identify them after being stored, so the correct HINTERNET handle, the one returned by the function above it in the hierarchy, can be passed to the current function call. To store these internet handles and other resources in a way they can be uniquely identified and referenced, a dictionary, or hash map, is created for each hierarchy level and type of resource. The key in the key-value pair is a string representation of the resource's unique hex or integer value used to reference the resource in the application. The value in the key-value pair is the actual reference to the resource object that will be passed to functions as an argument. The process for utilizing this data structure follows. The hooked malicious process sends the string representation of the resource parameter to the sacrificial VM server. The server compares this string to the keys stored in the associated resource's dictionary. The matching key maps to the desired resource object reference, which has been identified and is passed as an argument in the subsequent function call. Using this storage and reference process, the sacrificial VM can establish multiple network connections or allocate multiple file handles and perform the operations associated with those as facilitated by the malicious process.

4.3.2. Other Challenges. A few other notable challenges were encountered during the implementation of the diversion process and sacrificial VM server. First among those, the VM server must continue running and listening despite any exceptions or failed API calls. If the server hangs or quits unexpectedly, the hooking code running in the context of the malicious process will fail to receive the API call's return value, and the process typically terminates as a result. Because the goal is to keep the malware running in this hooked state, this behavior is undesirable. To keep the server running, all exceptions must be handled gracefully and any blocking calls, such as Winsock's `recv`, must be made non-blocking prior to their execution.

The remaining, notable obstacles met during system implementation have not yet been resolved. Though an assumption made during system design was that the malicious process had been identified, the assumption did not limit the process identification to the moment it was loaded into memory. As a result, the malicious process may execute for a period of time before the DLL is injected and IAT hooks put in place. The process execution before IAT hooking can lead to problems, especially if network connections are established in this time. If this is the case, the sacrificial VM will be unable to replicate the network connection establishment, and any subsequent operations to be performed over this connection will fail on the VM.

The last obstacle is created by a compiler optimization that occurs on occasion. If an API function is called multiple times within a section of code, for example as a part of a loop, the compiler sometimes moves the address of this function to a register. Using this method, the program can simply jump to the address stored in the register to call the function rather than looking up the address in the IAT repeatedly. If this optimization occurs before our IAT hooks are put in place, the compiler will store the original address of the API function in a register to be used in each call. When each call is made, the function address will not be looked up in the IAT; it will be acquired from the register, so our IAT hook will be useless as the address of our function will never be used. Fortunately, this compiler optimization has not been encountered often during initial testing of the AMC system.

4.4. Malware Execution

Employing the diversion process and strategies described above, the hooking code running in the context of the malicious process receives the return value from the sacrificial VM server and can manipulate it in any way or return it to the running malware unchanged. The malware then continues execution as though the Windows API call occurred as usual, not terminating unless by expected means. The server listens for a new connection attempt from the malicious process, triggered if another hooked Windows API call is encountered in the malware's execution. This is the workflow of the first version of the AMC system. As a result of this workflow, the mission-critical system should be able to continue running with a reduced impact from the running malware as the identified malicious functions are not executing in the mission-critical environment, but the dedicated VM environment.

5. Testing and Future Work

This section describes how initial testing has been performed and how tests can be improved. It also describes additions to the AMC system that would improve its effectiveness.

5.1. Testing

The components of the AMC system include DLL injection, IAT hooking, hook functions, and the sacrificial VM server. Testing of each unit and all units integrated together has been performed. Initial unit and integration testing verifies the functionality of the system, but it does not verify the effectiveness of reducing damage inflicted on the mission-critical system and the system's resulting longevity.

Testing has begun to evaluate how well the AMC system completes its primary objective. The most effective method developed for this assessment is to run malware and observe how the diversion process affects the malware's impact on the mission-critical system [19]. Assessments of this nature have been performed with several APT1 malware samples including Sword and Greencat [20] [21]. The AMC system successfully implements IAT hooks on imported, targeted functions and diverts their functionality to the sacrificial VM. Consequently, any operations that would be performed on the mission-critical system are instead executed on the sacrificial VM, and the critical environment experiences a reduced impact.

Though testing has been effective with many of the samples chosen, there have been a few shortcomings with other samples due to the testing environment. A dedicated testing environment has not yet been developed for running "live malware" with an internet connection. "Live malware" is used here as malware that is still being operated by its author; the malware is still capable of receiving communications from its command and control server and has not been de-weaponized in any way. Because many malware families depend heavily on communication over a network, typically the internet, to receive commands and send results, multiple samples terminate shortly after execution begins due to a lack of network connection, and the majority of functionality is unable to be tested without debugging and patching code.

5.2. Future Work

Several additions can be made to the AMC system to improve its effectiveness including adding more Windows

API hooks, supporting more versions of API functions, implementing in-line hooking, and testing the system extensively. The diversion stage's effectiveness relies solely on the proportion of hooked functions to total number of functions used maliciously. Identifying more Windows API functions utilized by malware to commit harmful acts and implementing hooks for those will reduce the negative impact on the mission-critical system that much more. In addition to IAT hooking, other hooking techniques could also be implemented to be utilized when IAT hooking is unavailable (e.g. if the malware dynamically links its libraries and builds its IAT at run-time using the `LoadLibrary` and `GetProcAddress` API calls). An alternative hooking technique is in-line hooking, described briefly in the hooking section of this paper [22]. Finally, the last improvement to the AMC system can be made in evaluating the effectiveness of the diversion on nullifying malware's effect on the mission-critical system. There does not currently exist a dedicated testing environment for testing the system with live malware. Creating this environment and performing tests with various malware samples will provide a more formalized assessment of the system's ability to accomplish the primary goal, keeping the compromised mission-critical system running normally.

6. Conclusion

All in all, this paper proposes the development and implementation of a system taking the unconventional approach at cyber-defense, minimizing damage after compromise has occurred. Named the Active Malware Countermeasure system, the software system's main objective is to limit the effect malware has on a mission-critical system to enable it to continue operating. The system diverts potentially malicious functionality away from the mission-critical system to a sacrificial VM. Diversion is achieved using hooking techniques, primarily IAT hooking, which is facilitated with code execution upon injection of a DLL into the malicious process's address space. The diverted functions are executed in the safe, virtualized environment, where any resulting damage can be wiped away using snapshots. Following function execution, the sacrificial VM returns data to the idle malicious process, so it can store and utilize this data in continuing execution. In this hooked state, the malicious process and the mission-critical system can both continue running but with a reduced impact on crucial operations.

7. Acknowledgements

This research was made possible in part by NPRP 9-005-1-002 grant from the Qatar National Research Fund (a member of The Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[1] W. McGrew, N. Sukhija, and D. Dampier, "Active malware countermeasure system," 2015.

[2] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16 – 24, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804512001944

[3] T. Osako, T. Suzuki, and Y. Iwata, "Proactive defense model based on cyber threat analysis," *Fujitsu Scientific and Technical Journal*, vol. 52, no. 3, pp. 72 – 77, 2016.

[4] "Clamav- open source antivirus system," https://www.clamav.net, 2017.

[5] G. G. Granadillo, J. Garcia-Alfaro, H. Debar, C. Ponchel, and L. R. Martin, "Considering technical and financial impact in the selection of security countermeasures against advanced persistent threats (apts)," in *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, July 2015, pp. 1–6.

[6] T. Zhang, H. Antunes, and S. Aggarwal, "Defending connected vehicles against malware: Challenges and a solution framework," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 10–21, Feb 2014.

[7] Y. S. Dai, Y. P. Xiang, and Y. Pan, "Bionic autonomic nervous systems for self-defense against dos, spyware, malware, virus, and fishing," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 1, pp. 4:1–4:20, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2567924

[8] H. Nakakoji, Y. Fujii, Y. Isobe, T. Shigemoto, T. Kito, N. Hayashi, N. Kawaguchi, N. Shimotsuma, and H. Kikuchi, "Proposal and evaluation of cyber defense system using blacklist refined based on authentication results," in *2016 19th International Conference on Network-Based Information Systems (NBiS)*, Sept 2016, pp. 135–139.

[9] S. E. Madnick and J. J. Donovan, "Application and analysis of the virtual machine approach to information system security and isolation," in *Proceedings of the Workshop on Virtual Computer Systems*. New York, NY, USA: ACM, 1973, pp. 210–224. [Online]. Available: http://doi.acm.org/10.1145/800122.803961

[10] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, "Subvirt: Implementing malware with virtual machines," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. SP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 314–327. [Online]. Available: http://dx.doi.org/10.1109/SP.2006.38

[11] M. Sikorski and A. Honig, *Practical Malware Analysis*, 1st ed. No Starch Press, 2012.

[12] M. Pietrek, "Peering inside the pe: A tour of the win32 portable executable file format," https://msdn.microsoft.com/en-us/library/ms809762.aspx, 1994, accessed: 18-Feb-2017.

[13] "Pe header and export table," http://www.cnblogs.com/shangdawei/p/4785494.html, 2015, accessed: 16-Feb-2017.

[14] N. Cano, "Import address table hooking," http://www.darkblue.ch/programming/Import_Address_Table_Hooking.pdf, 2016.

[15] D. Lukan, "Using createremotethread for dll injection on windows," http://resources.infosecinstitute.com/using-createremotethread-for-dll-injection-on-windows/#gref, 2013.

[16] "Certificate bypass: Hiding and executing malware from a digitally signed executable," https://www.blackhat.com/docs/us-16/materials/us-16-Nipravsky-Certificate-Bypass-Hiding-And-Executing-Malware-From-A

-Digitally-Signed-Executable-wp.pdf, 2016, accessed: 30-Jan-2017.

[17] "Windows msdn," https://msdn.microsoft.com/en-us/library/windows/desktop/ms632595(v=vs.85).aspx, accessed: 18-Feb-2017.

[18] "Hinternet handles (windows)," https://msdn.microsoft.com/en-us/library/aa383766(v=vs.85).aspx, accessed: 30-Jan-2017.

[19] R. Fanelli, "On the role of malware analysis for technical intelligence in active cyber defense," *Journal of Information Warfare*, vol. 14, no. 2, pp. 71–85, April 2015.

[20] "Contagio- malware dump," 2017.

[21] "Virusshare.com," 2017.

[22] B. Mariani, "Inline hooking in windows," https://www.exploit-db.com/docs/17802.pdf, 2011.