

Windows Packer/Loader

Mitchell Palmer   
*Cyber Security Engineering*  
*George Mason University*Fairfax, VA, USA  
mpalmer7@gmu.edu

Andrew Chapin  
*Cyber Security Engineering*  
*George Mason University*Fairfax, VA, USA  
achapin2@gmu.eduCarl Bai  
*Cyber Security Engineering*  
*George Mason University*Fairfax, VA, USA  
cbai2@gmu.edu

Andre Herrera  
*Cyber Security Engineering*  
*George Mason University*Fairfax, VA, USA  
aherre12@gmu.eduHunter Rowlette  
*Cyber Security Engineering*  
*George Mason University*Fairfax, VA, USA  
hrowlet2@gmu.edu

*Abstract*— Our project fulfills a request to produce a software toolkit that allows for remote code execution completely in RAM and file transfer via a service running on a remote host. The goal of our stakeholder, Lockheed Martin Corp. (LM), is for our research to identify a unique way to accomplish this task. The following requirements were provided by LM: (1) the toolkit must be comprised of two separate executables – a “packer” and a “loader”; (2) the “packer” runs locally on Linux, compresses, then encrypts with AES via a user-provided password before sending data to remote hosts; (3) the “loader” runs on a Windows remote host as a service, receives incoming packed data, decrypts/decompresses, and executes any PE files entirely in RAM (i.e. without touching disk). Other loader operating systems were desired. We delivered. A four-part concept of operations was established: (1) a user selects a data block (e.g. executable file) and sends it to the packer where it is packed, (2) the now-packed data is sent over the internet to the remote host, (3) the remote host receives the packed data with the running loader service, (4) the loader decrypts the data block and will either run it in RAM or make it available on the Disk. Specifically, a CLI was built for the packer for user interaction, and a heartbeat process for the loader was designed in order to communicate uptime and availability of remote hosts to the user. Our toolkit, written in C++, implements the desired objectives of our stakeholder. We utilized libssh, filesystem, miniz, etc. libraries to accomplish the objectives. Finally, quality assurance was established through integration and unit tests of the toolkit. The software was then handed over to LM for confirmation and testing in their environment. Alterations were made as requested and the final product was shipped. This paper provides an in-depth analysis of our product and our research into similar products and methodologies to our solution.

Keywords— cybersecurity, remote code execution, CLI

# Introduction

There is no best way to covertly and securely deliver a binary file to a remote computer system and have it executed without ever existing on the hard disk. Many ways will work - conditionally. Code that is functional for one operating system (OS) may not work on a different OS. Personal security products (PSPs) may catch one method of code execution, but not others. These factors depend on the security and configuration of the target remote host (RH). Networking factors can also play a role. Firewall restrictions, IDS/IPS, and network segmentation all affect the ability to deliver code. The method for securing network delivery must also be chosen wisely and is target specific. Our stakeholder, Lockheed Martin, asked us to tackle this research question and to deliver a two-part software product with specific conditions. The below paper explores the various methods for (1) secure network delivery of a payload and (2) remote code execution completely in memory; including a description of our product and how it achieves this task.

# Remote Code Delivery

## SSH Port Forwarding

For delivery of the file we decided to using SSH port forwarding. We set up an SSH server on the Windows Loader side, that the Linux Packer connects to. We opted to use SSH to help manage the key store. We utilized the functions included in libssh to create the SSH session between the Packer and Loader and then created a second channel for the port forwarding. This forwards any communications we want between the ports we assigned that packer and loader. Doing this allowed us to write the file to the channel and send it securely and discretely between the two.

## SSH Delivery (SCP)

SCP allows for files to be sent in a secure and encrypted way through the network between remote systems. SCP quite literally stands for secure copy and adds on to the functionality of the default Linux cp, or copy command. Unfortunately for this project, SCP does not include any support for communicating with non-Unix based machines by default. In order to achieve this functionality, additional software such as CYGWIN must be utilized.

## Metasploit Stagers

Traditional methods of malware deployment in the Metasploit framework include payloads known as singles, stagers, and stages. Singles are simply self-contained payloads that can be run on the target system. Single payloads are standalone and as such, they can be managed with non-Metasploit handlers such as Netcat. Stagers are smaller payloads that setup a network connection between the target victim and the attackers host machine. Once a stager is properly setup on the target, it can accept the final payload type; stages. Stagers are payload components that are downloaded by the stager's modules to produce something similar to a traditional single payload. An example of a stager would be Metasploit's reverse\_tcp with an example stage of the Meterpreter payload. Although stagers with stage malware tend to be small and reliable, they must be carefully crafted to work with one another. As such, most standard executables cannot be immediately supported as viable payloads. This is where a more traditional payload downloader/loader comes into play [4].

# Code Execution in Memory

Code execution that never touches the Disk is quite advantageous. It is much harder to trace back and analyze a program that has only ever existed in volatile memory which is wiped after the system is powered off.

## Methods for Execution in Memory

As the deployment base for this software is primarily windows-based clients, the methods for memory execution for binary blobs are limited. Depending on the exact nature of the binary to be executed and the restrictions on execution on the target system, one of two methods will be used. If the memory protection is looser, as is the case on some unhardened systems, data will be read into the memory as a binary array, and then the protection on that memory region will be swapped from a data to a code region. Then, a fork command will be called with a pointer to the beginning of that memory location. Alternatively, the certutil function can be pointed at a memory location and, when combined with other commands, can be made to execute arbitrary code in memory.

# Product Description

Our tool contains two binary executables dubbed   
“packer” and “loader”. They are written in C++. The packer is meant to be put on the attacker/deployer/local host and sends data to loaders located on remote hosts. See Figure [1]

## Packer

The packer is a C++ program with its own CLI and is built to run in Linux. It takes two flags: “-h” for the help menu and “-s” to run it as a “service”. When the packer is run in service mode, the user enters the CLI and is given four options: (1) list available payloads, (2) add a new payload, (3) send a payload, or (4) exit. Payloads by default exist in the “Payloads/” directory in the root folder for the packer. This can be changed to any path by editing the provided config file. When selecting “add a new payload”, the packer will import all payloads in the “Staging/” directory in the root folder. Likewise, this can also be changed to any path in the config. Adding a payload will compress and encrypt the payload with AES-256 via a user-provided password. "Send payload” will list all available payloads and prompt the user to pick one. Once selected, the user will have the option to select one or multiple target IPs that are running the loader service. The program will then attempt to send the payload over SSH and let the user know if it was successful.

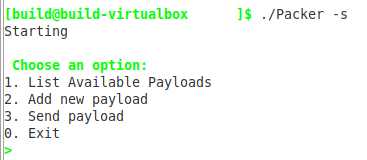


Fig. 2. Packer selection menu

## Loader

Our loader is also written in C++ and will run on Windows operating systems. Once a user has access to a Windows system, they can deploy the loader binary and set it to run as a Windows service (information on how to do so is provided in the loader “readme”). The loader utilizes the .NET framework to listen for incoming traffic over a specified port for data delivery from the packer. Once the loader receives data from the packer, it will decrypt the data with the same AES-256 key and decompress the data. The loader will then examine the magic bytes of the data to determine if the provided data is executable. If so, it will not save the data to disk, but execute it completely in memory. If it is not executable, the loader will save the data to disk, by default in the root folder of the loader.

## Future Development

Although our tool functions as both the downloader and loader for the malware, traditionally a packer/loader tool works in junction with a downloader. The malware is packed into a portable executable as an encrypted/compressed blob with a prepended loader stub designed to unpack it. This packed binary is then sent to the remote target and staged by a downloader or stager (the initial payload executed on the remote system). Once on the remote system, the downloader executes the loader stub on the packed binary to decrypt/decompress the binary and then pass the execution back to the original malware. Both the networking and packing of malware are complicated tasks that must consider changing security technologies and the state of the target operating system architecture. Developing a universal solution to perform all tasks perfectly is practically impossible and instead development should focus on small pieces of each problem. An ideal future design for an effective packer/loader solution would revolve around modular toolkits that perform small tasks but very effectively, that can then be combined and interchanged when needed to provide an extensible and adaptable form of deployment for malware.

##### Acknowledgment

We would like to provide special thanks to our stakeholder, Lockheed Martin (LM), and our primary point of contact at LM: Allison Elfring. Ms. Elfring was incredibly knowledgeable, professional, and helpful as the subject matter expert on the project. We would also like to thank Rock Sabetto for being everything we could have possibly wanted in a faculty mentor and George Mason University as our degree program.

##### References

1. “libssh 0.9.3,” libssh. [Online]. Available: http://api.libssh.org/stable/. [Accessed: 16-Mar-2020].

[2] R. Geldreich, “richgel999/miniz,” *Github*, 09-Mar-2020. [Online]. Available: https://github.com/richgel999/miniz. [Accessed: 16-Mar-2020].

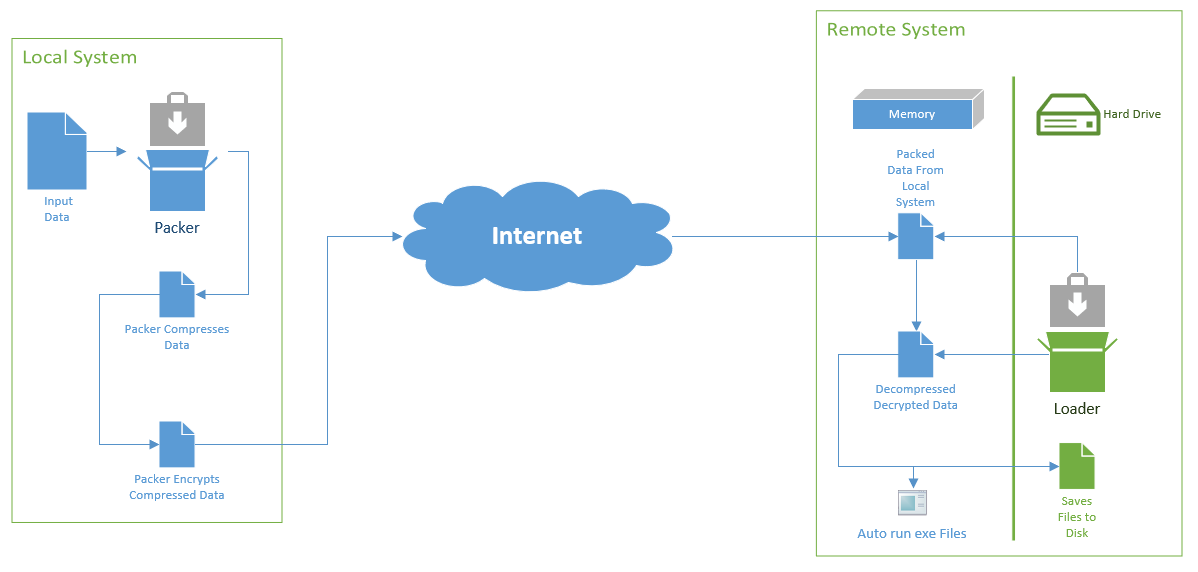
[3] “Filesystem library,” *cppreference.com*, 15-Jun-2018. [Online]. Available: https://en.cppreference.com/w/cpp/experimental/fs. [Accessed: 16-Mar-2020].

[4] “Payload Types in the Metasploit Framework,” PAYLOAD TYPES IN THE METASPLOIT FRAMEWORK. [Online]. Available: https://www.offensive-security.com/metasploit-unleashed/payload-types/. [Accessed: 16-Mar-2020].

[5] arno0x0x, “Windows oneliners to download remote payload and execute arbitrary code,” Windows oneliners to download remote payload and execute arbitrary code, 18-Apr-2018. [Online]. Available: https://arno0x0x.wordpress.com/2017/11/20/windows-oneliners-to-download-remote-payload-and-execute-arbitrary-code/. [Accessed: 16-Mar-2020].

[6]

Figures



[Fig. . Visual depiction of product functionality]

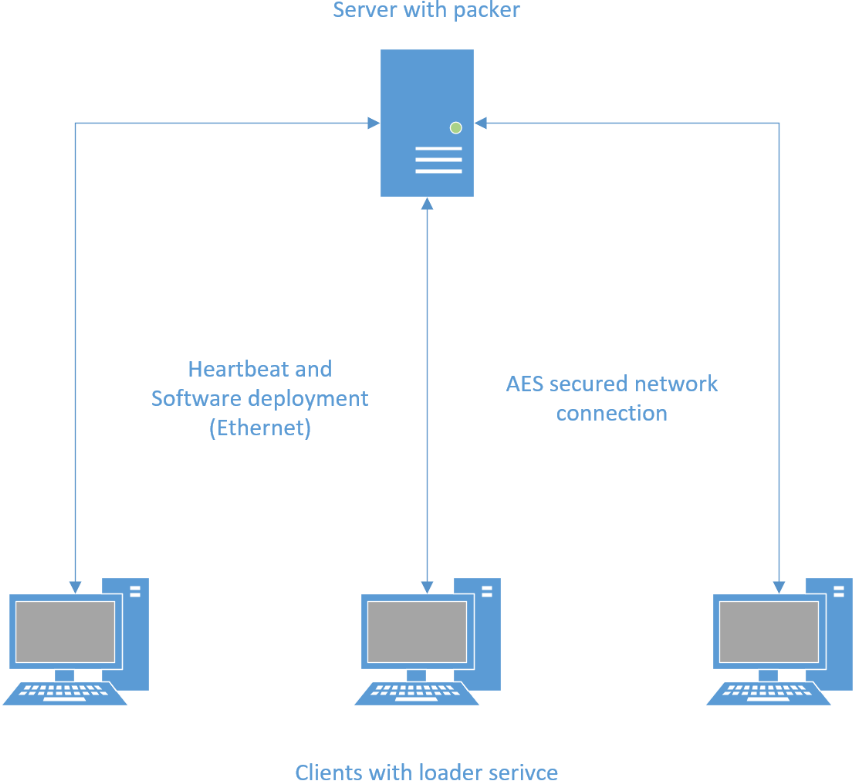


Figure 3 - Architecture Diagram