Alpha 3 Final Report

Camden Kronhaus & Pujan Paudel

Motivation

Our project's goal was to reduce the communication routes malware can utilize. Various malware use direct IP connections to communicate with servers. This includes P2P botnets that communicate with other bots, which is generally done through direct IP connection as opposed to registering every bot with a domain name, as well as many centralized botnets that communicate with a single C2 server. There are also malwares that may specify DoS attacks on specific servers that the malware has no control over assigning a domain name. Other malware that connects to a server, potentially via direct IP, are ransomware to report infection and payments, keyloggers and web injectors to report data, and other malware that may attempt to steal data or otherwise report its status in the infected system. Our goal was to mitigate malware such as these by forcing programs system-wide to resolve domain names before initiating communication with an IP address. By doing this, our project aimed to mitigate the strategies that malware could employ. This would increase overhead for malware developers to maintain DNS records for their internet infrastructure and thus facilitate tracking malware by gaining information from the domain names they would be forced to implement.

Anatomy of DNS Resolution on Linux

Our defense mechanism was built using Ubuntu 18.04 using the default network configuration that came during installation. The only additional mechanism we added later to the network configuration was adding IPv6 support so our system could also be run and tested with IPv6 addresses. The default configuration for Ubuntu uses systemd-resolved as its default system resolver. Through this configuration, systemd-resolved provides three interfaces it recommends using when resolving hostnames throughout the system. The first, most versatile and recommended way, is by using ResolveHostname(). ResolveHostname is a method that can be accessed by using the sd_bus_call_method(), giving it the parameter ResolveHostname, communicating with systemd-resolved over the Linux D-bus, providing network name resolution by forwarding DNS requests from nameservers specified by the servers. By default, these were the internal network IPs of the system. The second was using glibc's getaddrinfo(). Getaddrinfo() utilized the /etc/resolv.conf file within the Ubuntu system, reading it to connect to a resolver. With systemd-resolved this file is symlinked to /run/systemd/resolve/stub-resolv.conf, which maintains the only name server as localhost:53. Localhost:53 itself is part of systemd-resolved as it contains a local stub listener which connects it to the systemd-resolved service, and provides it with the DNS addresses maintained by itself. The last method provided by the systemd-resolved is direct IP communication. This connects requests to systemd-resolved as was just described using localhost. There are other methods that a program could use as well. DNS resolutions specifying a nameserver bypass localhost using Bind9 utilities, but we chose to focus on the most common/recommended methods for a program to use. The reason for this is, by way of our system design that we will describe, all communication bypassing hostname resolution through one of "our" functions is blocked by default. Therefore, the functions we focus on are primarily to allow as many "good" programs, ones that should be using the recommended approaches to resolving hostnames, as possible the ability to function normally.

malWALL Defense System

We propose a system, called **malWALL**, to force domain name resolution in Ubuntu. Our system is initially configured by applying firewall rules, using iptables/ip6tables, to block all outbound traffic not sent through port 53 or port 22. We allow port 53 for DNS queries and port 22 for SSH. If a process makes a DNS query using ResolveHostName() or getaddrinfo() then our system checks if the query is

done using a hostname, or directly using an IPv4 or IPv6 address. If it is done with an IP address, we return an error code and the firewall remains unchanged. As a result, the process cannot move forward with the connection attempt that it tried in the first place. When it is a legitimate DNS query resolution using a hostname, we proceed with normal execution and update the firewall with the returned IP. Once our system is in place, processes cannot establish any mode of connection with an IP address unless the hostname associated with it was resolved prior. By this somewhat rigid, but highly effective system, we try to limit the damage that can be done by malwares through direct IP connections.

Implementation

i) Initial exploration of tools

We first began designing our system trying to determine the best method for detecting when a program attempts to make a DNS query. We tried to determine what process makes a DNS query and what hostname it was trying to resolve. Then we tried to determine ways to detect IPs associated with specific queries. From the beginning, we avoided using a tcpdump-style approach, whereby we would read all the packets being sent out by the system, and attempt to filter by IPs that were resolved by communication through port 53. So, we began by attempting to utilize the utility SystemTap. SystemTap provided many methods for investigating a Linux system including function hooking and network filtering. We created a network filter using SystemTap but began having trouble developing further with it. The process of figuring out the best method for tracking IP requests got complicated with SystemTap, and we realized we were starting to default back to the tcpdump-style we were avoiding. Then the idea was brought up to attempt to use LD_PRELOAD, which ended up being integral to our system.

ii) LD_PRELOAD

LD_PRELOAD is a feature provided by the dynamic linker that can be used to overload C library functions. By using this functionality, we can tell the linker to bind symbols provided by a certain shared library before other libraries. When any program is executed, the operating system's dynamic loader will first load dynamic libraries linked into the process's memory so that the dynamic linker can then resolve methods at run time and bind them to actual definitions. We can modify the behavior of system libraries that are dynamically loaded (which is a strict requirement) by specifying the LD_PRELOAD environment variable with path(s) to a shared object that contains one or many methods called by the program being run. When the dynamic linker runs any executable, the program would instead call our specified method rather than the one found in standard library implementation.

Therefore, for any program that calls a method that we target to preload, we hook those methods to run our custom code before the actual function. Using this method, we could easily configure our approach to blocking all outbound network traffic utilizing IPtables (except for port 53 for DNS queries and port 22 for us to ssh into our development server). Once our specified methods are hooked system-wide, we would be able to track all DNS queries made using those methods systemwide.

iii) GetAddrInfo() method

We began by focusing on developing a preloaded getaddrinfo() wrapper. We investigated the parameters the method takes, and figured out how hostnames were passed in and how IP addresses were returned. We then split our wrapper into 3 steps. The 3 steps of operation is a common theme across the second library we preload, as well as any other libraries that could be preloaded in the future. The first step, pre-processing where we first ensure the hostname passed in is not an IP address, except localhost/127.0.0.1/::1. While we saw examples of programs call getaddrinfo() using localhost and therefore allow it to pass, we wanted to prevent any other IPs that malware could use to circumvent our system by attempting a false resolution of an IP address to the same IP address, a method that

getaddrinfo() allows and performs no actual lookup to resolve the address. We then perform the same method used by getaddrinfo() to resolve the hostname and gather the IP addresses returned. We loop through each IP address and for each call a C program that runs as root that simply adds an IPtable rule to allow outbound traffic to the IP address returned. Therefore, once a hostname is resolved, the IP becomes available system-wide for use, a result that doesn't weaken our system as the IP is certified to be resolved. The last step is returning an actual function call to getaddrinfo(), allowing the program to resolve and take in the hostname itself, afterward connecting to the IP the hostname resolves to. We found many network utilities that use getaddrinfo(). These included: ping/ping6, curl, wget, whois, C bind() and connect() methods, Python socket methods, and importantly the Firefox web browser, thereby allowing full use of a web browser so long as only hostnames are used by the user. As this is the normal with the current glibc method, this result makes sense. We tested with all of these and found our system to allow normal communication with all, and blocked all of them if they attempted to use direct IP connections (e.g. ping 8.8.8.8). We also found that certain commands like sudo also happen to connect to getaddrinfo() to resolve localhost while performing its operations. While we weren't sure why this was, our system performed no differently due to our implementation.

iv) ResolveHostName() method

We then expanded our work to preload ResolveHostname(). There was much less information on ResolveHostname() and we didn't find any utilities that explicitly used it, so we used an example given by freedesktop.org to communicate with it. This method was more convoluted as it couldn't actually be called directly and had to be called through a method that sends a d-bus message such as sd_bus_call_method(). sd_bus_call_method() is a convenience function for initializing a bus message object and calling the corresponding D-Bus method. The Linux D-Bus provides message-oriented middleware that allows concurrently running processes to communicate. We investigated this method's return values and parameters as we did with getaddrinfo(), and decided to preload the entire method. We first add a pre-processing step to determine if the message was addressed to ResolveHostname(). If such is the case, then the destination parameter of the method call is set to be "org.freedesktop.resolve1" and the member parameter is set as ResolveHostName. This filtering was necessary since the sd_bus_call_method could be used potentially by other functionalities on the system, which we didn't want to interfere with. Once the instance of method execution fulfills the categories, we investigate the additional parameters passed to execute ResolveHostname(). After that we integrated the same method as above, wrapping the method with our own that resolved the hostname using ResolveHostname() over sd_bus_call_method(), and adding the IP addresses it returned, before returning to normal execution.

iv) Direct DNS queries

We then investigated direct DNS queries. While we did work on attempting to listen to port 53 once again using SystemTap, and attempting to read the payloads sent, and the IP addresses resolved, we realized there were some problems with this approach. Besides the complications that reading network packets on the buffer level would bring, this would also introduce the problem of DNS hijacking. Whereas before, a program would have to use the system's DNS servers to resolve hostnames, by allowing direct requests, malware would be able to create a fake DNS server and return malicious IP addresses for seemingly normal domain names, bypassing our system. We observed a couple of utilities that used Bind9 methods to resolve hostnames either by default querying localhost:53 or if a nameserver was specified, querying that name server directly. We thought of different methods we could use to approach this and found that it would be possible to filter packets and force them to resolve hostnames using nameservers specified by systemd-resolved, thereby blocking the ability to specify nameservers entirely. But, we then observed that the utilities that were doing this type of DNS queries were all hostname resolver utilities. These included: host, nslookup, and dig. We realized that because all these

utilities did was perform a DNS query and not attempt to communicate with the resolved IP, by doing nothing we allow for these utilities to work properly, and any other program that solely wants to resolve a hostname, and does not compromise our system as our IPtables rules are not updated.

v) ***Putting it all Together***

After finishing the previous steps, we investigated a couple of areas to make our system better. First, we wanted to allow our program to run by a normal user, not on root. Second, we needed our program to run system-wide. Lastly, we wanted to test on real malware to show that our concept works to mitigate some kind of malware. To allow both running our programs as a normal user and systemwide, we found /etc/ld.so.preload. Ld.so.preload works similarly to LD_PRELOAD, but /etc/ld.so.preload is searched by all programs that run on a Linux system before running. By adding the file so that it exists, and putting the path to our shared object files that wrapped ResolveHostname()/sd_bus_call_method() and getaddrinfo(), any program on the system, even if run by a non-root user, would use our methods. If any program called the methods we hooked, it would call our functions and initiate the process of adding ALLOW rules to our IPtables. We did find that using ld.so.preload segfaults sshd. Unfortunately, despite investigating the crash reports, we were not able to solve this issue before our project deadline. At this point is where we added IPv6 support and added rules for ip6tables and checks for IP6 addresses.

Application: Blocking Malware connections

We collected the malware samples from a static repository made publicly available by VirusSamples.com[1]. The malwares are made available as Linux ELF binaries. We then proceeded with a basic static analysis of the malware samples. From the list of malwares, we used the utility command ***"strings"*** to extract all the static strings present in the malware. After getting the strings, we filtered out further by only considering strings that match the pattern of an IPv4 address. We further filtered out local IP and gathered a list of malwares that had at least one IPv4 address present in the static analysis of their binary. We ran a subsample of the relevant malware on a throwaway server running Ubuntu 20.04.2.0 LTS. In retrospect, using a throwaway server for malware analysis was not a good idea, and we should have invested some time in configuring a VirtualBox with the appropriate environment and network settings to run a more systematic and safer analysis. We executed each of the malware samples for three minutes and monitored the network traffic associated with the IP addresses of the malware samples individually. Only very few samples connected to the IP addresses within the three minutes of our monitoring time. We found 4 samples out of the 50 samples that made a connection to the IP addresses. Before our system is in place, the malware samples can easily establish TCP connections to their remote servers as we observe them on the network traffic. But once our system is configured, the malware's connection with their remote servers is blocked, and they cannot further proceed with their activity; which could range from sending back sensitive information to downloading payload and waiting for remote instruction to wreak more havoc.

[1] https://github.com/MalwareSamples/Linux-Malware-Samples

What we learned

We learned how to implement a defense system on a system-wide level and the plethora of nuisances that come along with it. First, we learned that the anatomy of DNS resolution in Linux systems is not as simple as it might look. There are multiple routes in the system that a DNS query might take and those routes change depending on the system configuration. We also learned all about the systemd-resolved methods and how they specifically resolve hostnames. Our defense system attempts to take care of the primary mediums of DNS requests, but addressing 100% of the possible methods is a really difficult task that would require in-depth knowledge of systems. We found LD_PRELOAD as a very powerful tool that has a lot of potential in developing defense systems on a diverse range of problems. Systemtap was also a very promising tool we encountered, but we found it to be rather limited on the level of functionalities it could offer. When we tried to inspect the traffic data on port 53, it had limitations on allowing us to read buffer data and other attributes of the network connection. Overall, this was a fun system to build and it was interesting to learn about system-level DNS queries, an area often overlooked.