EXAMINING THE SECURITY OF LOCAL INTER-PROCESS COMMUNICATION

Brendan Leech

Adviser: Professor Peter C. Johnson

A Thesis

Presented to the Faculty of the Computer Science Department

of Middlebury College

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Arts

May 2019

ABSTRACT

Your abstract goes here.

ACKNOWLEDGEMENTS

Your acknowledgements go here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Many modern applications split functionality into multiple processes, allowing programmers to achieve the design principle of separation of concerns. For example, by creating a password manager that uses two processes, one to store the passwords and a second to display them to the user, programmers can focus on the two very different tasks. One group can ensure that the stored passwords are unable to be stolen from the machine, while the other can provide the interface between the user and his or her information. This also allows someone with security expertise to only work on keeping the passwords safe and a user-interface designer to create a functional UI, instead of having one group work in areas that are not their strengths. However, separating related operations into different processes implies that they must communicate in some way. The password manager needs some way of getting the passwords from the secured database to the UI for the user to view. This is an example of inter-process communication.

1.1 Inter-Process Communication

Inter-process communication is any form of communication between running processes. This is a very broad definition that applies to much of the way that we use computers. Inter-process communication, or IPC, captures everything from email to using a web browser to get webpages from a web server to password managers transmitting passwords within a computer. Local IPC is communication that occurs within a single computer. Instead of traversing the internet and interacting with many hosts, local inter-process communication stays completely inside one computer, and is often dealt with entirely by the kernel. Inter-process communication and local IPC will be further explored in Chapter 2 Section 2.3.

1.2 Insecurity of Inter-Process Communication

1.2.1 What does it mean to be insecure?

For an application to be insecure, an attacker must be able to exploit the application to act in a way that is not desired. These attacks can be broken down into two broad classes: information disclosure and execution hijacking. An information disclosure attack is when an attacker gains access to victims' confidential information. This can include passwords, social security numbers, bank credentials, or other private information. These attacks have been in the news recently as large companies like social media giant Facebook [24] and credit agency Equifax [38] have been hacked and millions of users' private information was taken.

Execution hijacking attacks occur when an attacker is able to run arbitrary code on a victim's machine. If the attacker can run code as the victim, then he or she can act while pretending to be the victim. To an outside viewer, seeing a program running as a user on that user's computer would be expected, so attackers would be able to impersonate the victim without being easily spotted as a hacked machine. These attacks violate a victim's security and privacy by allowing hackers to steal confidential information and use a victim's identity to perform arbitrary actions. These two classes of attacks, while different, are similar in their lasting effects. Attackers are able to act as the victim and do what they would like. Whether that means using a stolen social security number to open a new credit card account or running a process on a victim's computer to send spam emails, the attacker gains new opportunities to act with the direct consequences falling onto the victim.

1.2.2 Security of Networked IPC

Since it requires other computers to handle a user's data, networked inter-process communication is fundamentally insecure. Any computer on the route between the source and destination is given the data, and in theory, could do whatever it wants with the data. This could include storing the data and attempting to decrypt the data offline, or monitoring the traffic that different hosts and users send. Without encryption, any computer along the path between the source and destination computers would be able to read all information passed in the message, allowing information disclosure vulnerabilities to be trivial. This would include passwords, credit card numbers, and other forms of private information that are constantly sent through the internet. To protect against this, much of the confidential information sent across the internet is encrypted. As the internet has become more popular, more and more information is being encrypted when sent over the network. In the past, HTTPS, the encrypted version of HTTP, was used for secure transactions only, such as entering a credit card number to make an online purchase or typing in a password to log into an account. However, HTTPS is used more than ever before [1]. This allows users to protect their browsing history and helps to reduce the ability of attackers to forge website URLs through the use of certificates. However, even with this and other precautions, anytime that personal or confidential information is sent through other machines, that communication should be considered insecure.

1.2.3 Security of Local IPC

Local IPC, on the other hand, is completely contained within a single computer. The messages stay within the machine, and are almost always handled by the kernel itself. However, that does not mean that this communication is completely secure. In fact, since many believe that communicating within a single computer is secure, security precautions that are standard for networked communication are often missing in local IPC [7].

It is not the case that programmers do not try at all to secure this communication; in fact, there is often some form of security, but it is not enough, as shown by [7]. These researchers studied the ways that applications communicate locally and were able to impersonate the client or server, or both, in a dozen commonly-used applications. Further, they showed that it is possible to do so while also making it difficult for the victim to know that their machine has been compromised. Using the 'nohup' command on Macs and Linux and fast-user switching on Windows, users are able to keep programs running even after they have logged off. Most personal computers have only one personal account, but many users do not turn off the guest account. Using this guest account, an attacker could start a malicious program and log off. The victim would only know that his or her computer was compromised if he or she were to look through all of the currently running processes. An even more fertile ground for this attack is a public terminal. Many institutions such as libraries and universities have computers with many accounts on them. For example, at Middlebury College, every member of the faculty and staff, as well as each student, has an account. Therefore, each of these people is able to log onto the many public computers that the College makes available. By targeting these public terminals, an attacker would be able to gain access to the confidential information of many different users, instead of only one at a time on a personal computer.

1.3 Plan of this thesis

Since we know that local IPC is of concern, it would be helpful to characterize the security of individual applications that use this form of communication. Therefor, I first created a survey to see what commonly-used applications run on people's computers and what is each application's local IPC footprint. After gathering these results, I found the applications that are most used, as well as which applications used the most of each of the three forms of local IPC that I am investigating: communication over localhost,

UNIX domain sockets, and named pipes. This way, I looked at applications that are used everyday by many people and which use different forms of local IPC. I chose to look at APPLICATION1, APPLICATION2, and APPLICATION3 because of REASON1 and REASON2. Then, I took a two-pronged approach to investigate their security. First, I studied the communication between processes to see if it would be possible to send phony messages and hijack execution or access confidential information due to lax security procedures. Secondly, I used fuzzing software, discussed further in Section 3.4, to find cases where a process does not correctly parse its input. If a fuzzer's input could cause a crash in the process, then that would signal a bug that could possibly be exploited.

1.4 Related Work

1.4.1 Man in the Machine Attacks

In a 2018 paper, Bui et al. looked at the forms of local IPC used by common applications and found that they were able to read communication between processes and either hijack execution or disclose confidential information [7]. They named this attack the Man in the Machine Attack, since the attacker is using communication that stays within the computer.

These authors explored the situation where an attacker had access to the victim's host, but neither as an administrator nor as the victim. Instead, the attacker used a separate login session, either as another authenticated user to that computer, or using the guest account. Many people who use a personal computer do not disable the guest account, which leaves their computer open to possible attacks. Additionally, a public terminal with many accounts, such as at a university or an office, would allow an authenticated user to possibly steal confidential information from many people, instead of

the single victim of a personal computer. Using fast user switching on Windows [21] or running a program with 'nohup' on MacOS and Linux allows a program to run even when the user who started the program is no longer logged in, or the user is running in the background. Using these techniques, an attacker could start the malicious program while logged into an account, then log out while continuing their attack.

Bui et al. looked specifically at three vulnerable types of local IPC: network sockets, Windows named pipes, and universal serial bus (USB) security tokens. Network sockets provide an easy attack vector for both client and server impersonation. Since the authors only investigated local IPC, the IP address for all communication was the localhost address, 127.0.0.1. Therefore, they only needed to find what port the software used to find the communication between the processes. A server listens on a set of specific port numbers that is defined in the source code, so a malicious process would need to connect to one of these ports to impersonate a client. If the server can only support one connection, then the malicious client must connect before the genuine client does. To impersonate the server, the malicious program must listen on the selected ports before the actual server has the chance to bind to them. By creating a fake client and then a fake server, an attacker is able to complete a man-in-the-middle attack.

Using named pipes, it is similarly easy to impersonte the actual named pipe. Instead of using port numbers, named pipes have a name, or location in the filesystem, that can be used to identify them. To impersonate the client, the malicious program must join the named pipe as a reader, and to impersonate the server, the program must create the named pipe before the real server does. The only requirement for either of these is to know the name of the pipe, which can easily be found by running the 'lsof' program on UNIX operating systems or the 'handle' or 'pipelist' programs on Windows [28] [18]. The attacker can then use this name for his or her attack.

The last class of vulnerable local IPC is USB security tokens. A USB device is

available to any user once it is plugged into the machine, so the programmers behind the software on the USB must implement security features to protect unwanted use of the device. Without these features, any user, including the malicious user, could access the USB token.

Bui et al. also found that there are some types of local IPC that are immune to the man in the machine attack. The two most common types are anonymous pipes and anonymous socket pairs. Anonymous pipes are often used in pipelines while using a shell. A command like 'ls | grep *sys*' uses an anonymous pipe to take the output of 'ls' and send it as input to 'grep'. These pipes are safe while named pipes are not because the anonymous pipe is created by one process, so it owns both ends of the communication channel. The pipe also does not have a name that can be joined by other processes. Therefore, for another process to have access, it must be given one end. This mostly occurs between parent and child processes, as in the shell example above. Socket pairs are safe for the same reason. Since the sockets do not have names and cannot be joined by outside processes, any process that uses them is considered safe because it must be explicitly granted access by the process that owns the sockets.

The authors went on to study four classes of applications: password managers, USB hardware tokens, applications that have an HTTP backend, and two other applications of interest. Of the thirteen applications studied, twelve were vulnerable to some form of impersonation, while the last was vulnerable to signing incorrect two-factor authentication requests.

Some of the studied password managers had such careless security that a man in the machine attack was trivial. RoboForm connected its browser extension with the password database through localhost, and communicated the password in plaintext. To steal a user's passwords, the attacker only needed to connect to port 54512 on localhost, ask for a list of accounts, and then choose one of the keys that was sent from the database.

The database would then send the password associated with that account in plaintext. While the other password managers had stricter security, Bui et al. were able to impersonate one side of the conversation because of weak key-exchange protocols, secret keys stored in Javascript code, or other easily fixable security holes.

The two applications that used hardware tokens are used for two-factor security. While the password is the "thing you know," a physical two-factor device is a "thing you have." Two-factor authentication with a security token is employed when a user wants extra security for an account by requiring both a "thing you know" and a "thing you have." For example, one of the studied tokens, Fujitsu DigiSign, is used by the Finnish people to interact with government services, including healthcare resources [7]. The attack on this token takes the primary port that the card-reader listens on before the real client can, then impersonates the server by connecting to the real client from a secondary port. This allows the attacker to have malicious requests signed by the token and the user.

The other applications, while lacking the security that is required of a password manager or security token, still have insufficient security. For example, Spotify is a music streaming platform that uses a localhost socket to play music. After connecting to the server, an attacker can spoof the Origin field in the header of the HTTP request so the malicious request will be accepted by the server. They could then design the payload of the packet to change the song that the victim is listening to. MySQL is a database server that can be configured to use named pipes. Here, an attacker can join the server's real named pipe as a reader and create its own instance for the client to join, resulting in a man in the middle attack. The attacker can then query the database itself, as well as modify legitimate queries to and responses from the database.

1.4.2 Local IPC Vulnerabilities

Bui et al. were not the first to identify vulnerabilities in local IPC. Many have found issues with the way local IPC is implemented by kernels. Named pipes, specifically in Windows, have well-documented security issues. First of all, the default access rights for a Windows named pipe allow anyone to read it, meaning that a programmer must change the default to securely use a named pipe [20]. Additionally, a lack of complete understanding of how to use named pipes created many vulnerabilities in Windows applications that gratuitously used named pipes, including a remote code execution attack against qBittorrent [9]. Additionally, by exploiting the way that Windows creates named pipes, a named pipe writer could gain the security context of and impersonate the reader [39].

However, named pipes are not the only form of local IPC with vulnerabilities. It has been shown that apps on both iOS and Mac OS X can use IPC to gain access to all the resources of a victim app [40]. A malicious app was able to bind to the port used by the 1Password password manager before the real app could. Then, the attacker could steal passwords, even though the two apps were supposed to be sandboxed from each other. On the Android operating system, UNIX domain sockets have been shown to be vulnerable as well. A malicious program could give itself root access, view the confidential files of other apps, or access the Bluedroid (Android Bluetooth) radio and control devices connected through Bluedroid [34]. These vulnerabilites are due to a lack of authentication by both processes connecting to the socket.

CHAPTER 2

INTER-PROCESS COMMUNICATION

Inter-process communication (IPC) is the way that any two processes communicate with each other. Anytime that two processes need to comunicate, they use IPC. This can include different applications, such as a web server sending pages to a web browser, or separate processes of the same application, such as Spotify and Spotify Helper working together to play music. When this communication occurs within one computer, it is local IPC.

2.1 Benefits of IPC

There are many problems that are well-solved by having a multi-process application. These are normally problems that have multiple distinct functions, since each task can be separated into its own process. Examples of applications that fall into this category would be web servers, password managers, and XWindows.

WEB SERVERS DISCUSSION - NEED SOURCES FOR THIS

Password managers also make use of multiple processes to separate the jobs of storing passwords securely and entering the password into forms. A password manager may have three processes running at all times. One process encrypts and secures the passwords on disk, ensuring that no malicious user can access the plaintext of the passwords. Some of these processes go so far as to create fake sets of passwords, so that an attacker would not know which set of encrypted passwords is real [5]. A second running process could be an app on the computer so that a user can read or edit the password, and a third process could be a browser extension to autofill passwords into webpages. Both of these processes need to communicate with the password storage process so that they can receive the passwords. The browser extension needs the plaintext of the password before it can put it into the webform, as does the desktop app before it can display the

password to the user. The communication occurs via local IPC, since all three processes would run on a single host.

Password managers lend themselves to having multiple processes because all three processes solve different jobs. The password vault needs to keep the passwords encrypted and secure from outside access, but needs to provide the password when either of the other two processes legitimately requests it. The desktop app should provide an easy-to-use user interface so that the user can edit or view the desired password. Finally, the browser extension needs to find password fields in online forms and automatically fill them with the correct password based on the URL of the webpage. By splitting each task into its own process, each process can be optimized for its use. As long as the passwords can be communicated between the processes, they will be able to work together and function as a successful password manager.

XWindows is another application that benefits from using multiple processes. XWindows is a program that displays the graphics on a monitor. It uses two processes, a client and a server, to create and present the images. The server process takes in input from the mouse, keyboard, and other peripherals and sends it to the correct client, while also receiving information from the clients about what should be displayed [31]. Each client process is a different application which takes in mouse and keyboard data, does computation using this and the current state of the application, and sends to the server what it would like shown on the screen [31]. Using this model, one server can have many clients connected to it. This allows a single screen to display multiple applications at the same time, since each application has its own client. Also, this means that the server can demultiplex incoming signals from the mouse and keyboard and send them to the correct client. This architecture also allows a single application to be displayed on multiple screens, since a client can connect to multiple servers. XWindows would almost certainly be unable to achieve the same benefits if it was a single-process application.

The benefits gained from using multiple processes cannot be replicated with a single process.

2.2 Local IPC Background

Local IPC is inter-process communication that occurs entirely within a single computer. Both password managers and XWindows, described above, utilize local IPC often. Many password managers exist completely within one computer, so they only communicate the passwords within that host. The XWindows client and server are often run within a computer as well. In these scenarios, they use local IPC to efficiently communicate and work together. Local IPC also has the benefit of avoiding some of the overhead required for networked communication since it is guaranteed to stay within the computer. This can make local IPC significantly faster than networked communication. The tradeoffs of using different forms of local IPC will be discussed further in Section 2.3.

Since local IPC never leaves the host, the security implications of the communication are less clear than networked IPC. With networked communication, the messages go through other people's computers, making the communication intrinsically insecure. However, since local IPC is contained to one computer, some security expers believe it is not worth trying to encrypt this communication and keep it secure. They believe that if an attacker has access to the host, which would be required to exploit an application that only uses local IPC, then he or she already has full access to the computer, so any effort to encrypt or keep information secure is futile. Section 3.1 has more context about host-only attack vectors. As shown by [7], many applications do make some effort to encrypt local IPC, but not to the same extent as their networked communication. They use defective key-exchange protocols or lack two-way authentication to confirm the other party they are talking with.

2.3 Forms of Local IPC

This thesis studies three forms of local IPC: communication through localhost, UNIX domain sockets, and named pipes. We will discuss each in more detail here.

2.3.1 Localhost

Localhost, also known as the loopback interface, is an interface provided by computers to communicate with itself. This interface can be used to replicate the complete network stack, and is especially useful when testing networked applications without a working internet connection. However, this interface is also used for local inter-process communication. One of the benefits of using localhost for local IPC is that the entire infrastructure used for networked communication can stay the same. The only change that needs to be made is to use the IP address of localhost, 127.0.0.1, instead of the IP address of another interface.

When communicating over localhost, a message is sent using the entire network protocol stack, including the link, network, and transport layers. Each layer contains a header, which provides metadata about that layer, and data, which can be any assortment of bits. For example, IP is a protocol in the network layer, so sits in between the link and transport layers. An IP packet has a header that contains the source and destination IP address, a checksum, and a field that describes the type of information contained in the data section. Then, in the data section, an entire TCP or UDP packet would be contained, which would hold the TCP or UDP header, as well as the data inside of that layer. This process is called encapsulation, since each higher layer is encapsulated as data within the next outer layer.

The link layer is used to differentiate which computer on a physical wire should accept incoming frames. A normal ethernet frame contains the source and destination

MAC addresses, each six bytes long, as well as two bytes to indicate the network layer being used. However, when using localhost, this can be optimized, since only one computer, the current host, is on the "wire" and it should listen to all incoming connections to localhost. Therefore, when using localhost, the entire link layer header is four bytes that determine the family of network layer. Most of the time, this will be IP, which has the family value of 2.

The network layer used when communicating over localhost is the same as is used for networked IPC. This layer is predominantly IP. The IP header contains information about what interface the packet is destined for, as well as what type of data is contained inside [25]. While the ethernet layer can be condensed when communicating through the loopback interface, the IP layer cannot, because the kernel needs to know what interface to send the packet to, which is represented in this layer. The kernel has to route the packet to the code to encapsulate the IP packet as either an ethernet or localhost frame, so must be able to know the destination IP address.

The last layer commonly used in the network protocol stack is the transport layer. This layer contains port numbers to identify the source and destination process, as well as sequence numbers to guarantee delivery of messages, if the transport layer is TCP [26]. The common transport layer protocols are TCP, which sends a stream of data, and UDP, which sends individual frames, called datagrams. TCP also adds a guarantee of delivery and in-order delivery, which UDP and the network layer protocols do not have. Like the network layer, this layer is needed because the kernel needs to know, for both ends of the communication, what process to give the packet to.

Inside of the transport layer is the application layer. This can be a popular protocol like HTTP or the bit torrent protocol, or it can be a proprietary protocol. Many applications have their own application layer protocol to transmit the exact information that is needed, along with the desired amount of security.

When a process uses the loopback interface to communicate with another process, it uses this entire protocol stack: the condensed link layer, the network layer, and the transport layer. This can be very useful for a programmer, since from a programming perspective, the code is the exact same as the code for networked communication, except for the destination IP address. There is no need to use different data structures or macro values, since the communication almost completely mocks real internet communication. Also, if there is a possibility that the communication endpoint will change from localhost, it is very easy to modify the code since only the address would be changed.

2.3.2 UNIX Domain Sockets

If the programmer knows that the communication will never leave the computer, then he or she may decide that the overhead of the entire network stack is not necessary. Therefore, the programmer could instead use UNIX domain sockets, a form of local IPC that works similarly to internet sockets. Like internet sockets, UNIX domain sockets, also known as UNIX sockets or IPC sockets, create a bidirectional channel of communication. These sockets can be created to send a stream like TCP, datagrams like UDP, or can be a raw socket. However, raw sockets are almost never used [35, 229–230]. While internet sockets use an IP address and port number as the namespace to find and send packets, UNIX domain sockets use the filesystem as the namespace [35, 231]. However, while the namespace is different, the same commands are used to create, write to, and read from both internet and UNIX domain sockets.

In addition to the lower overhead than internet sockets, UNIX domain sockets have two additional benefits that no other form of IPC can replicate. UNIX domain sockets are able to send file descriptors and credentials to the connected socket [36, 381–394]. By being able to send file descriptors, UNIX domain sockets provide a way for processes to share file descriptors outside of fork and exec. Additionally, this is allowed for any

type of descriptor, so processes can send pipe, socket, or file descriptors through a UNIX domain socket. Once the descriptor is sent, the receiver will be able to open it whenever it chooses, even if the sender closes the descriptor before the receiver opens it. The other unique benefit of UNIX domain sockets is their ability to pass credentials through the socket as well. This can be used as a security check by a server process to guarantee that a client is allowed to request the service to be done. This is the only way to guarantee that a process is receiving the genuine credentials of a client.

To send data across a UNIX domain socket requires many fewer steps than to send data via an internet socket. To send data over a datagram UNIX domain socket, the sockets are first connected using the destination pathname given. To send the data, control information, if given, along with the sender's address and the data itself are placed at the back of the receiver's receive queue by the kernel and processes waiting to read from the recieving socket are woken up [35, 263–265]. Control information would contain descriptors or credentials that are passed through the socket. The sender's address is not necessarily required, although the receiver will not be able to reply is the sender does not include its address; this can be ok in circumstances when the sender does not need a reply. Finally, the data is the bytes that the sender actually wants to send.

Just like with a datagram UNIX socket, sending a stream of data with a UNIX socket is much easier than over an internet socket. To send a stream of data, the kernel connects the two sockets if they are not already connected, then the data is moved to the receiver's receive queue [35, 265–268]. Any readers that are waiting for input on that socket are then woken up to read the incoming data, and the reader updates the size of the sender's and reciever's queues to reflect that the data has been read. The kernel is able to move the data directly from the sending process to the receiving process with just the required permission checks.

2.3.3 Named Pipes

Sockets that use the loopback interface and UNIX domain sockets provide bidirectional communication channels, but that is not always necessary. If only a unidirectional channel is required, then processes can open a named pipe. A named pipe, or FIFO, is a special type of file that lets a process send data to another process. Named pipes use the filesystem as their namespace. A pipe's 'name' is given when it is created, and this is used when another process wants to connect to the pipe. Once a pipe exists in the filesystem, any process that knows, or guesses, the name of the pipe can open one end, either as a reader or as a writer. Using a named pipe looks as if the named pipe is a normal file and is being written to and read from using output and input redirection. However, a named pipe is more efficient than storing the data in a temporary file since the kernel can buffer it instead of writing it to disk.

Named pipes are different than anonymous pipes, since they are able to be joined as long as a process knows the name. However, anonymous pipes, like those used in pipelines in a shell, have much stricter access controls. When an anonymous pipe is created, only the process that created it can gain access to it. Often, this process will fork and exec soon after, which allows the child process to have access to the pipe as well. In this sense, a process must be explicitly given access to an anonymous pipe, either through UNIX domain socket descriptor passing or as a child to a process with the pipe open. This is very different from named pipes where any process is able to open either end.

2.4 Tradeoffs Between Forms of Local IPC

When deciding what form of local IPC to use, all three of these types: using the loopback interface, UNIX domain sockets, and named pipes, provide benefits that should be

considered by application programmers. By using the full network stack and the loopback interface, programmers can use the exact same commands and arguments that they are used to using from creating networked communication. The only difference is that the destination IP address will always be the loopback interface. If they decide to use localhost, they then must decide whether to use TCP or UDP in the transport layer. TCP requires more overhead, such as the three-step-handshake to create the connection, but guarantees delivery and in-order delivery. However, a programmer may be confident that packets will very rarely be lost by the kernel, since they never leave the host, and could want the lower cost of UDP.

However, if the overhead of the network stack is too high for a specific application, a programmer could use UNIX domain sockets. UNIX domain sockets avoid almost all of the network stack, and the kernel transmits the data directly from the sender to the receiver. In fact, on four different Berkeley-derived systems, UNIX domain sockets were over twice as fast as TCP sockets that used the loopback interface [36, 223–224]. XWindows takes advange of this speed boost when it starts up by seeing if the server and client are on the same host, and if they are, creates a UNIX socket instead of an internet socket. UNIX domain sockets also have the ability to pass file descriptors and user credentials, giving other processes access to objects they previously could not access and giving them a way to guarantee that they are receiving genuine credentials.

UNIX domain sockets also can be created using the socketpair system call, which creates both endpoints and gives both endpoints to the creating process. This is similar to a process creating an anonymous pipe, since no other process has access to read from or write to the socket, unless it is explicitly given access by being sent a descriptor or through a fork. This process is so similar to creating a pipe, that anonymous pipes are often actually made by the socketpair system call, then one end is made read-only and the other is made write-only [35, 253].

Named pipes have a similar advantage to UNIX domain sockets where their namespace is the filesystem, so any process that knows their name can join as a reader or writer. They also provide a unidirectional data channel if that is desired by the application architecture. Named pipes do not have any inherent security, so programmers must be careful that a named pipe that they create is the first instance.

All of these forms of local IPC, except for UNIX sockets created by the socketpair command, must use some form of authentication to ensure the other end of the communication is the correct program. Any program is able to read or write to an internet socket, UNIX domain socket, or named pipe, so security measures need to be put in place by the application.

To decide the right choice of local IPC, programmers need to know how their application will be transferring data. As shown in [41], the size of writes can affect the speed of transmission. Therefore, depending on the size of data sent at a time, different forms of local IPC may be preferable. Based on [16] and [36], we can conclude that of the three forms of local IPC studied, named pipes are the fastest, followed by UNIX domain sockets, followed by sockets that send over localhost. These speed differences are significant, but other factors contribute to the decision of which form of local IPC should be used.

CHAPTER 3

SECURITY OF IPC

3.1 Attack Vectors agains Host-only Applications

As discussed in 1.2.2, networked IPC is innately insecure. Therefore, any application that uses the internet must take precautions to keep communication secure. However, applications that do not use the internet, host-only applications, have their own sets of attack vectors. The two most vulnerable attack vectors are memory leaks and local communication channels.

3.1.1 Memory Leaks

Memory leaks occur when a process does not flush memory and leaves confidential information dereferenced in memory. For example, if a password manager is in use, it will likely store passwords in memory. Once the user finishes using the password manager and locks it, the passwords in memory will be freed as the process is cleaned up. However, if the password manager does not scrub memory, for example by using 'strcpy' to replace the memory with 0s, then the next process to be given that area of memory could read the passwords.

This situation is not just a hypothetical. This year, it has been found that five common password managers, including 1Password and LastPass, fail to adequately scrub memory before it is freed [4]. While there are limits to what a password manager can do to keep passwords secure, the applications researched failed to reach them. The password that is being requested by a user must be in memory in plaintext while in use so that the client process is able to use it, but the password manager should scrub this region of memory immediately after the password is taken. However, applications like KeePass and LastPass fail to scrub any password after they are accessed the first time,

leaving them in memory in plaintext, even after the password manager is locked. Of even greater concern, 1Password 7 puts all passwords into plaintext in memory when the password manager is unlocked, along with the master password. An attacker who is able to read arbitrary memory would be able to find all of a user's password manager. These password managers, along with all applications that handle confidential information, should strive to scrub memory regions immediately when they are no longer needed to minimize the risk of data leaks.

This concept of scrubbing memory as soon as possible is not new. It was outlined in 2005 under the term "secure deallocation" [8]. Secure deallocation means that memory is scrubbed as soon as all processes are finished using it. At the time of this paper's publication, the lifetime of data was commonly from first write until the next time that data was written in the same location, regardless of whether a new process owned the data. The secure deallocation timeframe defines data living in memory from first write until it is explicitly freed, showing that it is no longer needed. The ideal lifetime would be from first write until last read, however this would be impossible for an operating system to know when the last read will be. By using secure deallocation, the operating system would be able to automatically scrub data as it is being freed, minimizing the time when confidential information is living in memory.

3.1.2 Communication Channels

SHOULD I SUMMARIZE THE EARLIER SECTION IN INTRO ABOUT INSECURITY OF LOCAL IPC? I DON'T WANT TO BE REPETITIVE

3.1.3 Other Attack Vectors

THIS SECTION IS ON MY LIST FOR NEEDING MORE SOURCES

3.2 Input Management and Parsing

When attacking an application, hackers often craft an input that executes the program in a way that the creators did not intend. This input may take advantage of lapses in the parsing algorithm of the vulnerable program. Parsing, or input management, is the way that a program decides whether input is correctly formatted, and if so, how to deal with it. For example, part of a compiler will be a parser that checks through the code to make sure that the programming language syntax is correct, such as balanced parentheses and semicolons at the end of lines. If there is a bug in the parser, then invalid input will be allowed into the program, possibly turning a small bug into an exploitable vulnerability. This invalid input can follow execution paths that were unintended to occur by the program and possibly take the program into a state that was not supposed to occur.

Therefore, if a programmer is able to create a provably perfect parser, then he or she will be able to remove the possibility of a large class of vulnerabilities: input–based vulnerabilities. These will be discussed more in-depth in the next section, Section **??**. The difficulty in creating a perfect parser largely depends on the complexity of the input being given. If the input language is too complex, then it will be impossible to prove that a parser is perfect.

To break down the problem more, we will call the set of all possible, valid inputs the input language. For a parser to be correct, for any given string, the parser must correctly decide whether the string should be accepted or rejected. If the string is accepted, then it is in the input language. Otherwise, it is not. If the input language is regular or context-free, then we can prove whether or not the parser accepts exactly the input language. If so, then we have created a perfect parser for our input language. In this case, we could place the parser at the beginning of the program, so that any input immediately goes through the parser. Accepted strings would be sent to the program to run, while rejected

strings would cause the program to end immediately, without the input ever reaching the actual program logic. With a perfect parser, we can guarantee that only valid input reaches the application and would be able to prevent input–based vulnerabilities.

However, many input languages were not designed with this in mind, so the langauge is at least recursive. Because of this, being able to prove that a parser only accepts the input language is an undecidable problem [30]. This is not necessarily because a specific input language needs to be recursive, but more because programmers do not explicitly think about the difficulty that the problem of parsing represents.

This problem is further complicated by the way that parsing logic is currently implemented in many pieces of software. In many applications, programmers use "shotgun parsing," which means that the parsing logic is spread out throughout the code, instead of doing all parsing at one time [6]. Since the parsing code is spread out, it is more difficult to check that all possible cases are covered, even if the input language is decidable. Another trap that programmers fall into is using a regular expression in an attempt to validate an input language that is not regular [6]. In this case, whether an input should be accepted could be decidable, but the logic used to determine this does not have enough computational power to do so.

To combat these weaknesses, a design philosophy called Language Theoretic Security, or LangSec, has risen in popularity. LangSec follows the idea that the code that decides whether input is valid should be separate from the application code that processes the input [2]. In a LangSec-compliant program, once the application process receives the input, it knows the exact form that the input will follow, without exceptions, and therefore can operate without any need to check for input correctness. This helps to make the processing code cleaner since there will be no need for ad-hoc validity checks. More importantly, the application will be much safer since the program will be safe from a large class of exploits.

3.3 Input-Based Vulnerabilities

When an application's parser does not correctly accept the input language, the application is left vulnerable to input-based vulnerabilities. These are vulnerabilities where the attacker crafts and uses input that breaks some of the assumptions that the program writers have made to make the application perform unexpectedly. These are especially powerful for hackers because, if the exploit succeeds, they get to run a program of their desire on the victim's computer [30].

When an input-based vulnerability is exploited, the program will not behave in the way the programmer intends. For example, the program could switch execution to another program, often a shell, to give the attacker the ability to perform arbitrary execution on the victim computer. Another common attack uses holes in the parsing logic to investigate memory that the programmer does not want to be retrievable by users.

While these attacks are dangerous when done in user-space, they are even more effective when the vulnerable software is a system call. A system call is the way that a process can interact with the hardware of the computer, either by writing to the screen, reading or writing to a file, and many more important tasks. Unlike user applications, which run in the least privileged protection ring of a computer, system calls run in the most privileged ring, often called ring 0. When in ring 0, the program has access to all memory, not just the program's own memory. Therefore, when bugs exist in system calls, the vulnerable system calls can be used to get any information existing in memory or to hijack execution with the most privileges. When a user passes parameters to a system call, the process running with higher privileges is vulnerable to cause a kernel panic, change user permissions, and many other consequences [17].

With this in mind, we will look at three different classes of input-based vulnerabilities: buffer overflows, format string attacks, and other attacks that violate assumptions.

3.3.1 Buffer Overflows

Buffer overflows are one of the most common exploits, earning the nickname "Vulnerability of the Decade" for the 1990s [10]. Even twenty years later, buffer overflows are still very common because of how many buffer overflow bug exist as well as the power they give an attacker. A buffer overflow occurs when an attacker puts more bytes into a buffer than it can hold, overwriting memory after the buffer. Buffer overflow attacks became common after a paper entitled "Smashing the Stack for Fun and Profit" was published in 1996, describing how one could overwrite the return address of a function, jumping execution to another location in memory that the attacker could have previously filled with their own instructions [23]. While an easy fix to this attack would be to check the bounds of input before writing it to a buffer, this is often forgotten, or the programmer may believe the bounds were previously checked by parsing logic. These bugs are especially common in software written in C because there is no built-in bounds checking; it must be implemented by the programmer. Many protections, including stack canaries, non-executable stacks, and ASLR have been implemented by kernels to reduce the possibility of buffer overflows, but all of these can be defeated [27] [33] [13]. In some cases, either for efficiency or due to their age, programs may be compiled without a stack canary or with an executable stack, allowing simple buffer overflows to be effective.

One example of a buffer overflow attack existed in libpng version 1.2.5 [11]. Using a malformed PNG image, an attacker could overflow a buffer for transparency data and cause arbitrary execution. Since libpng was and is still used for much of the internet's PNG handling operations, any place where this version of libpng was used was vulnerable. If an attacker could get the malicious PNG file to be ran through a specific function, then he or she would be able to run any desired code on the victim's computer, including opening a shell. If the victim compiled their code without a stack canary and with an

executable stack, then the attack would be even easier. This shows the power of a buffer overflow attack and the widespread damage that can be done.

3.3.2 Format String Attacks

Another type of input-based vulnerability is a format string attack, which occurs when an attacker adds placeholders to a formatted string input. For this to occur, the programmer must choose to not use a format string, but instead make the user-inputted string varaible the only argument to printf or other string formatting function. In these situations, an attacker can view or overwrite memory to either disclose information or hijack execution [22]. Another use of format string attacks is to overwrite memory addresses in the global offset table [32]. The global offset table, or GOT, contains addresses for all of the library functions called. Then, when a function is used, execution jumps to either the runtime linker or the function itself, depending on if the function has been previously linked in the running program. By overwriting this address, an attacker can change execution to any arbitrary address, without worrying about the protections against buffer overflow attacks.

However, while format string attacks were once a fertile ground for vulnerabilities, they are also an opportunity to be a success story of correct parsing. In C, the syntax for format strings is regular, so it is possible to create a provably correct parser to make sure that there are no additional placeholders in the user's input [29]. However, since the attack occurs when the user inputs the format string, this cannot be parsed because the format string often is supposed to have placeholders. It should not have placeholders when there are no additional arguments supplied to the function, so this is an attack that needs a redesign of the printf and associated functions as well as additional parsing.

3.3.3 Other Violated Assumptions

The last class of inout-based vulnerabilities is the general class of other ways that assumptions can be violated. These vulnerabilities, as do the ones previously discusses, occur when the application developers do not think of the ways that applications could be attacked. For example, SQL injection attacks occur when user input is given directly to a SQL query, allowing users access to the database without the input being cleaned. This attack is so prominent that it was rated the biggest application security risk of 2017 [3]. SQL injection attacks include attempts to extract data, modify or destroy databases, or avoid authentication [15]. These can be mitigated by checking the types of inputs, searching for correct input patterns, and using intrusion detection systems. However, SQL queries are not regular nor context-free, so using regular expressions does not correctly separate valid and invalid inputs, and there is no provably correct parser for all inputs.

While SQL injection attacks have no complete solution, other vulnerabilities do. The Heartbleed bug, which was announced in 2014, occurs when an attacker abuses the heartbeat extension in OpenSSL [19]. The heartbeat extension works when one side of the connection sends a payload and its length, and the other side is supposed to send the same payload back. The response works when the payload is moved into memory, and the responder replies with the same number of bytes as identified in the payload length field. However, there is no check to make sure that the length field is no longer than the actual payload length. This allows an attacker to specify a much longer length field, which returns the bytes of memory after the original payload, up to the payload length field [12]. This bug occurs because the programmers expected the specified payload length to agree with the actual length of the payload, but did not actually check that they did [6]. This simple mistake highlights the high risk of input-based vulnerabilities. It is easy to overlook many of these vulnerabilities and their consequences can be devas-

tating. This underlines the need to identify all input-based vulnerabilities if there is no way to make a provably correct input parser.

3.4 Fuzzing

One way to go about finding input-based vulnerabilities is called fuzzing. Fuzzing is the process of sending random, semi-random, or unexpected input to a process [37, 21–22]. The goal is to find inputs that cause the application to hang, crash, or otherwise behave unexpectedly. This could represent a bug in the parsing code, where some aspect of the input is not being handled correctly and is causing problems downstream in the application. Often, developers will fuzz their own applications before shipping to find and eliminate as many bugs as possible. However, fuzzing can be done by third-parties as well, either to improve the software or find bugs to exploit. This type of fuzzing is more difficult since it is impossible to ensure that every execution path has been covered without the source code [14].

CHAPTER 4

CHAPTER 3

CHAPTER 5

CHAPTER 4

APPENDIX A

CHAPTER 1 OF APPENDIX

Appendix chapter 1 text goes here [if you have one...]

BIBLIOGRAPHY

[1] Https encryption on the web.

[2] Langsec: Recognition, validation, and compositional correctness for real world security.

[3] Owasp top 10 - the ten most critical web application security risks, Mar 2018.

[4] Password managers: Under the hood of secrets management, Feb 2019.

[5] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant password management. In *European symposium on research in computer security*, pages 286–302. Springer, 2010.

[6] Sergey Bratus, Lars Hermerschmidt, Sven M Hallberg, Michael E Locasto, Falcon D Momot, Meredith L Patterson, and Anna Shubina. Curing the vulnerable parser: Design patterns for secure input handling. *USENIX; login*, 42(1):32–39, 2017.

[7] Thanh Bui, Siddharth Prakash Rao, Markku Antikainen, Viswanathan Manihatty Bojan, and Tuomas Aura. Man-in-the-machine: Exploiting ill-secured communication inside the computer. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1511–1525, Baltimore, MD, 2018. USENIX Association.

[8] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security Symposium*, pages 22–22, 2005.

[9] Gil Cohen. Call the plumber: You have a leak in your (named) pipe, Mar 2017.

[10] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.

[11] Multiple buffer overflows in libpng 1.2.5 and earlier. National Vulnerability Database, November 2004.

[12] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and

J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[13] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press, 2016.

[14] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[15] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[16] Patricia K Immich, Ravi S Bhagavatula, and Ravi Pendse. Performance analysis of five interprocess communication mechanisms across unix operating systems. *Journal of Systems and Software*, 68(1):27–43, 2003.

[17] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, volume 2, page 0, 2004.

[18] Markruss and Kent Sharkey. Pipelist, Jul 2016.

[19] Neel Mehta and Codenomicon. The heartbleed bug, Apr 2014.

[20] Microsoft. Named pipe security and access rights, May 2018.

[21] Microsoft Developers Network. Fast user switching, May 2018.

[22] Tim Newsham. Format string attacks, Sep 2000.

[23] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[24] Donie O'Sullivan. Facebook's worst hack ever could get worse, Oct 2018.

[25] Jon Postel. Internet protocol. STD 5, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc791.txt.

[26] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt.

[27] Gerardo Richarte et al. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[28] Mark Russinovich. Handle, Dec 2018.

[29] Len Sassaman, Meredith L Patterson, Sergey Bratus, Michael E Locasto, and Anna Shubina. Security applications of formal language theory. *IEEE Systems Journal*, 7(3):489–500, 2013.

[30] Len Sassaman, Meredith L Patterson, Sergey Bratus, and Anna Shubina. The halting problems of network stack insecurity. *USENIX; login*, 36(6):22–32, 2011.

[31] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Trans. Graph.*, 5(2):79–109, April 1986.

[32] Team Teso Scut. Exploiting format string vulnerabilities, March 2001.

[33] Hovav Shacham et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security*, pages 552–561. New York,, 2007.

[34] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z. Morley Mao. The misuse of android unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 80–91, New York, NY, USA, 2016. ACM.

[35] W. Richard Stevens. *TCP/IP Illustrated (Vol. 3): TCP for Transactions, HTTP, NNTP, and the Unix Domain Protocols*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[36] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1997.

[37] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[38] Craig Timberg, Elizabeth Dwoskin, and Brian Fung. Data of 143 million americans exposed in hack of credit reporting agency equifax, Sep 2017.

[39] Blake Watts. Discovering and exploiting named pipe security flaws for fun and profit, 2002.

[40] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os x and ios. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 31–43, New York, NY, USA, 2015. ACM.

[41] Zhang Xiurong. The analysis and comparison of inter-process communication performance between computer nodes. 2011.