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

1 Introduction 1

1.1 Inter-Process Communication 1

1.2 Insecurity of Inter-Process Communication 2

1.2.1 What does it mean to be insecure? 2

1.2.2 Security of Networked IPC 3

1.2.3 Security of Local IPC 3

1.3 Plan of this thesis 4

1.4 Related Work 5

1.4.1 Man in the Machine Attacks 5

1.4.2 Local IPC Vulnerabilities 9

2 Inter-Process Communication 10

2.1 Benefits of IPC 10

2.1.1 Web Servers 10

2.1.2 Password Managers 11

2.1.3 XWindows 13

2.2 Local IPC Background 14

2.3 Forms of Local IPC 15

2.3.1 Localhost 15

2.3.2 UNIX Domain Sockets 18

2.3.3 Named Pipes 21

2.4 Tradeoffs Between Forms of Local IPC 23

3 Security of IPC 25

3.1 Attack Vectors agains Host-only Applications 25

3.1.1 Memory Leaks 25

3.1.2 Communication Channels 27

3.2 Input Management and Parsing 27

3.3 Input-Based Vulnerabilities 30

3.3.1 Buffer Overflows 31

3.3.2 Format String Attacks 32

3.3.3 Other Violated Assumptions 33

3.4 Fuzzing 34

4 Methods 35

4.1 Survey 35

4.2 Extracting Results 36

4.3 Fuzzing 37

4.4 Seeing Local IPC Communication 38

5 Results **40**
5.1 General Results 40
5.2 Applications 41
5.3 Spotify 43
5.3.1 UDP 43
5.3.2 TCP 44
5.3.3 UNIX Domain Sockets 45
5.3.4 Spotify Conclusion 45
5.4 VS Code 45
5.4.1 TCP 46
5.4.2 UNIX Domain Sockets 47
5.4.3 VS Code Conclusion 49
5.5 launchd 49
5.5.1 TCP 49
5.5.2 UDP 50
5.5.3 UNIX Domain Sockets 50
5.5.4 launchd Conclusion 51
5.6 mDNSResponder 52
5.6.1 TCP 52
5.6.2 UDP 52
5.6.3 UNIX Domain Sockets 53
5.6.4 mDNSResponder Conclusion 53

6 Future Work **54**

7 Discussion **55**

A Chapter 1 of appendix **56**

B Code Samples **57**
B.1 Writing to a Named Pipe 57
B.2 Reading from a Named Pipe 59
B.3 Loopback Interface TCP Server 61
B.4 Loopback Interface TCP Client 64
B.5 Loopback Interface UDP Server 67
B.6 Loopback Interface UDP Client 70
B.7 UNIX Domain Socket Stream Server 72
B.8 UNIX Domain Socket Stream Client 75
B.9 UNIX Domain Socket Datagram Server 78
B.10 UNIX Domain Socket Datagram Client 80
B.11 `socketpair` UNIX Domain Sockets 82

C Complete Data Table **87**

Bibliography **96**

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 division of labor 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 encompasses any form of communication between running processes. This is a very broad definition that applies to much of the way that people 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.

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 [29] and credit agency Equifax [44] 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 message. This could include storing the data and attempting to decrypt it 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. Now, HTTPS is used more than ever before [4]. 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 [11]. 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 [11]. 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, all of these people are 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. Therefore, 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 the loopback interface, 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 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 [11]. 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 [26] 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 address of the loopback interface, 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 a fake server, an attacker is able to complete a man-in-the-middle attack.

Using named pipes, it is similarly easy to impersonate either the client or server communicating through named pipes. 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 [33] [22]. 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 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 use of them is considered safe because another process 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 other 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 the loopback interface, and communicated the password in plaintext. To steal a user's passwords, the attacker only needed to connect to the loopback interface's port 54512, 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 a password is a "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 a password-protected 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 [11]. 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 protection guarantees that are required of a password manager or security token, still had insufficient security. For example, Spotify is a music streaming platform that used a socket connected to the loopback interface to play music. After connecting to the server, an attacker could 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 [25]. 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' [13]. Finally, by exploiting the way that Windows creates named pipes, a named pipe writer could gain the security context of the reader and impersonate the reader [45].

However, named pipes are not the only form of local IPC with vulnerabilities. It has been shown that malicious applications on both iOS and Mac OS X can use local IPC to gain access to all the resources of a victim application [46]. An attacker was able to bind to the port used by the 1Password password manager before the real app could. Then, he or she could steal passwords, even though the two applications 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 [39]. 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. This can include processes in 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 include web servers, password managers, and XWindows.

2.1.1 Web Servers

Web servers have the difficult job of serving thousands, if not millions, of connections at the same time. One method to deal with this is to use multiple processes, one for each of the new connections, and one parent process to accept new ones. The parent process starts listening on the desired port, and accepts every new connection that is made. Then, it creates a new process which is in charge of servicing a single connection. In this structure, each child concerns itself with only one client, and can serve data and webpages as fast as it can be scheduled. The parent is then left alone to accept connections without needing to worry about individual clients.

By keeping the tasks of accepting and servicing connections independent of each other, a web server can work more efficiently. If one process had to do both jobs, then it would accept a connection and service it until the client disconnected, and only then be able to accept a second connection. This means that only one connection would be allowed at a time, an unacceptable condition for a web server. Therefore, there needs to be some form of parallelism so that new connections can be accepted while also satisfying already connected clients. In this case, new processes create this paralellism, but a similar solution spins off a new thread for each connection. Both of these systems allow web servers to provide immense amounts of data to clients, much more than would be possibly by a single, single-threaded process.

2.1.2 Password Managers

Password Manager Usage

Password managers also make use of multiple processes to separate the different jobs it performs. Password managers are used to help make user's passwords more secure. It is difficult for users to come up with strong passwords since the recommendations for a good password–long, random assortments of letters, numbers, and special characters–are the same strings that are extremely hard for humans to memorize. It is impossible for users to memorize a unique, strong password for every single account they have. Instead, users will often choose weak passwords and repeat them over many accounts, since this lowers the cognitive burden. Password managers strive to make passwords more secure by remembering all but one of a user's passwords. All the user needs to know is the one master password to unlock the manager. Then, the user will be able to either read, copy, or autofill the password from the manager and into the application.

A password manager often has two separate abilities; it can display the passwords to the user or autofill them. To display the passwords, the manager will have an application designed to let users easily view and change their saved passwords. The application will open, and the user will be prompted to enter his or her master password. This will unlock the password vault, which is the process in the background that holds the passwords and sends them when a valid request is made. Once the vault is unlocked, the user will be presented with the different account names that are saved–a Google account, an banking account, and an ESPN.com account–for example. Once a user clicks on the desired account, the password will be fetched from the vault and shown to the user. There may also be a feature that would allow the user to copy the password, so as to paste it into a form later. When the user closes the application, the vault will be locked again.

The other way that password managers are often used is to autofill usernames and passwords into web browsers. They do this by using a browser extension. For example, when you download 1Password, you are able to download the desktop application and the browser extension. When using a browser extension, if the user travels to a webpage that has sign-on information, GMail for example, the user will be able to choose the username and password saved in their password manager for their GMail account. Then, the password manager will retrieve the username and password and autofill both into the webform. This way, users can easily enter their passwords on websites without ever having to leave their web browser.

Password Manager Processes

Because of the different jobs that password managers must accomplish, they are an ideal candidate to split the tasks into multiple processes. A password manager may have three processes running at all times. One process is the vault, which 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 [9]. A second running process is the desktop application to let a user read or edit their passwords, and a third process would be a browser extension to autofill passwords into webpages. These last two 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.

2.1.3 XWindows

XWindows is another application that benefits from using multiple processes. XWindows is a program that displays graphics on a monitor. It uses many processes, multiple clients and one 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 [36]. 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 to be shown on the screen [36]. 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.

In early uses of XWindows, the client computation would be done on a separate machine from where the monitor and peripherals were located. This allowed the machine the user interacted with to quickly demultiplex inputs from the user to the clients and multiplex the images from each process to the monitor. This would be done using a networked connection. Now, computer processors have enough power that these tasks do not need to be split onto separate computers, but can be accomplished on a single host. In this case, XWindows will no longer use the Internet to communicate, but instead will use local inter-process communication [42, p 373].

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 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 2.4.

With networked communication, the messages go through other people's computers, making the communication intrinsically insecure. However, since local IPC never leaves the host, the security implications of the communication are less clear. Some security experts believe that it is pointless to defend against local attackers, the same type of attackers who perform Man in the Machine attacks [11]. However, if software developers used the same best practices for key-exchange protocols, encryption, and two-way authentication that are used in networked communication, then there would be no difference between the attacks possible on networked and local IPC. Unfortunately, software has often been written without these standards extended to local IPC, creating the possibility for Man in the Machine attacks.

2.3 Forms of Local IPC

This thesis studies three forms of local IPC: communication through Internet-based sockets on the loopback interface, UNIX domain sockets, and named pipes. I will discuss each in more detail here.

2.3.1 Localhost

localhost is the name traditionally associated with the IP address assigned to the loopback interface: 127.0.0.1. Computers often provide the loopback interface as a way to communicate within themselves, using the complete network stack, without the messages leaving the machine. It 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 the loopback interface instead of the IP address of another, possibly remote, 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 addresses, 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 only a four byte field that determines the family of network layer. Most of the time, this will be IP, which has been assigned the value 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 [30]. 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 next 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 [31]. 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 which 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 “BitTorrent” protocol, or it can be a proprietary protocol. Many applications have their own application layer protocol to transmit the exact information that is needed and in a precise format. Encryption and other forms of security are also added at this layer.

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, the transport layer, and the application 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 to a remote interface, it is very easy to modify the code since only the destination IP address would be changed.

Code samples of a TCP server and client are given in Appendices B.3 and B.4. A UDP server and client are given in Appendices B.5 and B.6.

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 communication channel. These sockets can be created to send a stream like TCP, datagrams like UDP, or can be a raw socket. A raw socket allows the programmer to structure both the header and the payload for either the link or network layer, instead of allowing the kernel to construct the headers. In practice, raw sockets are almost never used [41, p 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 [41, p 231]. One benefit of this is that UNIX domain sockets do not need to use the network layer at all, since routing between machines is not necessary [23, p 753]. 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 other benefits that no other form of IPC can replicate: UNIX domain sockets are able to send file descriptors and credentials to the connected socket [42, p 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`. Normally, when a process has privileges that it would like another process to have, it will create a new process as a child. The child will then transform itself into a new process, often by using the `exec` system call. However, this requires the two processes to be related since one must explicitly create the other. UNIX domain sockets can be used between completely unrelated processes. 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. 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 performed. This is the only way to guarantee that a process is receiving the genuine credentials of a client through a UNIX domain socket [42, p 391].

Sending data across a UNIX domain socket requires the kernel to take many fewer steps than doing so with an Internet socket. To connect two sockets, which is required for stream sockets and optional for datagram sockets, there are checks to make sure that the pathname exists and that a socket of the correct type is bounded to that pathname. Then, for stream sockets only, when the client attempts to connect to the server, a second socket is created and added to the server's queue of incomplete connections [41, p 240–245]. After this, the sockets are connected. For stream sockets, the additional socket is moved from the incomplete connection queue to the completed connection queue [41, p 245–249]. Both types of sockets must be explicitly bound to receive a reply, because unlike Internet sockets, UNIX domain sockets do not implicitly bind when sending data.

To communicate over a datagram UNIX domain socket, 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 [41, p 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 if the sender does not include its address; this is acceptable in circumstances when the sender does not need a reply. An example of a datagram UNIX domain socket server and client are given in Appendices B.9 and B.10.

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 moves the data to the receiver's receive queue [41, p 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. An example of a stream UNIX domain socket server and client are given in Appendices B.7 and B.8.

`socketpair`

UNIX domain sockets, while often created with a name in the filesystem, can also be made to exist abstractly. This is done with the `socketpair` system call, which returns two connected UNIX domain sockets. The UNIX sockets do not have a name in the filesystem, so are similar to an anonymous pipe. In fact, anonymous pipes used to be made by calling `socketpair` [8] and then making one end read-only and the other write-only [41, p 253].

No other process has access to read from or write to either end of a `socketpair` connection, unless it is explicitly given access. This can be done by sending either descriptor through another UNIX socket or through a `fork` or other subprocess creation mechanism. By using `socketpair`, a process can be more sure that its communication is secure since that process is entirely in charge of giving others access. There is no way for another process to gain one of the socket descriptors without being given it from the owning process.

Without being given an endpoint, participating in the conversation over a `socketpair` socket is impossible. Additionally, while reading communication through the loopback interface is easy with applications like Wireshark [1], and any process can connect to a named pipe or named UNIX socket and send or receive communication, this is much more difficult with an abstract UNIX socket. There are a few ways to read the traffic, but all require extraordinary steps.

First, you could modify the kernel source code for the send functions to not only send the data, but also output the data somewhere to be read later. This will give all communication sent on the computer, but it will be difficult to find the desired messages as well as difficult to modify Mac source code and successfully reinstall it. The next option is to use `dtrace` or another tracing utility to track all of the send functions that a specific process calls. While this will only give you the communication for the desired process, you must disable System Integrity Protection to trace applications. System Integrity Protection protects against writing to system files and directories and many common malware attacks. Therefore, it is very dangerous to turn it off. A third solution is to use a proxy or other piece of software, like 'Unix socket sniffer' [2] to actively examine the data as it is transmitted. However, for this solution to work, you must be able to connect the two sockets yourself, either across a 'chroot jail' or by directing traffic to the proxy. For sockets that are already connected, this is not possible.

Therefore, because a `socketpair` socket does not exist in the filesystem, it is a very secure form of local IPC. Additionally, the extreme steps required to monitor traffic through a specific UNIX socket, even a named one, makes these anonymous sockets even safer. An example program that creates sockets using `socketpair` is given in Appendix B.11.

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 any process can join a named pipe as long as it knows the pipe's 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.

Named pipes have been implemented using UNIX domain sockets [40, p 1147]. When a FIFO is created, two UNIX sockets are created and connected, then one endpoint is made read-only and the other is made write-only. This enforces the unidirectionality of named pipes. This socket has type SOCK_STREAM, just like an Internet socket using TCP, so it is stream-oriented as opposed to a datagram-based connection.

Sample programs that write to and read from a named pipe are given in Appendices B.1 and B.2.

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 functions and arguments that they are used to using from 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 [42, p 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 [42, p 373]. 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.

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` system call, must use some form of authentication to ensure the other end of the communication is the correct process. Any process is able to read or write to an Internet socket, a normal UNIX domain socket, or a 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 [47], 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 [20] and [42], I 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 the loopback interface. These speed differences are significant, but other factors contribute to the decision of which form of local IPC should be used.

This chapter has addressed the three forms of local IPC that I am examining: Internet sockets that connect to the loopback interface, UNIX domain sockets, and named pipes. This has provided half of the background implied by the title of this thesis. The next chapter will provide the other half by looking at attack vectors for single-host applications, input management and input attacks, and finally a way to defend against input attacks.

CHAPTER 3

SECURITY OF IPC

3.1 Attack Vectors agains Host-only Applications

As discussed in Section 1.2.2, networked IPC is innately insecure because it uses other people's machines. Therefore, any application that uses the Internet must take precautions to keep communication secure, such as encryption and two-way authentication. 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 wipe its memory, for example by using `memset` to replace the memory with arbitrary data, 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 was found that five common password managers, including 1Password and LastPass, fail to adequately scrub memory before it was freed [7]. While there are limits to what a password manager can do to keep passwords secure, the applications researched failed to reach them. The password 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 passwords stored in the 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" [12]. 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.

While secure deallocation has a large security benefit, it also has downsides. The largest downside is that it takes extra time. Without secure deallocation, when a user denotes that an area of memory is no longer needed, all the kernel needs to do is put those memory addresses into the available memory pool. However, if the kernel needs to also wipe that memory, then that will make the free operation take more time and CPU cycles. Zeroing memory made processes run up to 10% slower, which is an unacceptable delay for any process where confidential data is not of critical importance [12]. This could also cause problems in processes that use memory after they have deallocated it. While the second issue is a bug resulting from poor programming, the first represents a significant slowdown that must be addressed before secure deallocation can successfully enter common use.

3.1.2 Communication Channels

While memory leaks attack information once it is no longer being used, attacks can also attempt to read or interfere with information while it is being used. In particular, as discusssed in Sections 1.2.3 and 1.4.1, local communication channels can be vulnerable. Many common applications, including security-conscious applications such as password managers and security tokens, were vulnerable to client or server impersonation due to weak encrpytion, key-exchange, or other vulnerabilities that could be easily solved. Additionally, other researchers have been able to use local IPC to break application isolation on both iOS and Mac OS X devices as well as hijack execution of the Android Bluetooth radio to control Bluedroid-connected devices. These attacks highlight the insecurity associated with local communication channels, even though the data sent will never leave the computer. It still must be secured as if it will traverse the Internet so as to protect users from local attackers.

3.2 Input Management and Parsing

All applications must handle input, and this determines the application's behavior. This input can come in terms of command-line arguments, such as what file the `cat` program should print the contents of, to mouse clicks that a web browser must translate into action. This input can also be text-based, such as a website letting a user input a term to filter data by or filling in an online form with personal information.

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. It infers meaning from raw bytes. For example, part of a compiler is 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 bug into an exploitable vulnerability. This invalid input can follow execution paths that were not supposed to happen by the program and possibly take the program into an unintended state.

Therefore, if a programmer is able to create a parser and prove that it accepts exactly the desired input, 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 3.3. The difficulty in creating a this 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 accepts exactly the appropriate input.

To break down the problem more, I 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 it is possible to prove whether or not the parser accepts exactly the input language. If so, then a parser for our input language has been created. In this case, one 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 this parser, it is guaranteed that only valid input reaches the application and the parser 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 [35]. 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 [10]. 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 [10]. 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 [5]. In a LangSec-compliant program, once the application logic 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 it 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 [35].

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 mode, 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 writing to the screen, reading or writing to a file, and many other 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 kernel mode. When in kernel mode, the program has access to all memory, not just the processes' 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 [21].

With this in mind, I 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 [14]. Even twenty years later, buffer overflows are still important because of their large quantity 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 were documented by a paper in 1996 entitled "Smashing the Stack for Fun and Profit" that described 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 [28]. 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 [32] [38] [17]. 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 [15]. 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 can occur what a function like `printf` is run. This can occur in many situations, such as when there is a mismatch between the number of format parameters and the number of parameter arguments, or when a specific format parameter is used, `%n`, which will output the number of bytes written to a variable. In these situations, an attacker can view or overwrite memory to either disclose information or hijack execution [27]. Another use of format string attacks is to overwrite memory addresses in the global offset table [37]. 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. This can also be used to cause a denial-of-service attack if the attacker can get the program to attempt to read memory it does not have access to, causing a crash [37].

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 placeholders in the user's input [34]. Since the user cannot add format parameters to the string, as long as the programmer correctly matches the number of format parameters with the number of parameter arguments, most attacks that attempt to read memory will fail. However, if the `%n` format parameter is still used, attackers would always be allowed to write arbitrary numbers to memory.

3.3.3 Other Violated Assumptions

The last class of input-based vulnerabilities is the general class of other ways that assumptions can be violated. These vulnerabilities, as do the ones previously discussed, 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 [6]. SQL injection attacks include attempts to extract data, modify or destroy databases, or avoid authentication [19]. 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 [24]. 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 moves the payload into memory, and the responder replies with the same number of bytes as identified in the payload length field. However, there was no check to make sure that the length field is no longer than the actual payload length. This allowed 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 [16]. This bug occurred 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 [10]. 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 devastating. 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 [43, p 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.

Fuzzing can be split into two other categories: whether or not the fuzzer is able to read the source code. With the source code, fuzzers can see control-flow structures such as if statements and loops and use these to follow all possible execution paths. In practice, this requires too much time and too many resources to follow every single execution path, but there is a notion of how many have been tested. In contrast, without any source code, it is impossible to ensure that every execution path has been convered [18]. Without knowing how the program is supposed to run and exactly what would cause different paths to be taken, the fuzzer has no idea how many possible execution paths are left untested.

CHAPTER 4

METHODS

4.1 Survey

To begin exploring how securely local inter-process communication is used, I first needed to find out what applications use local IPC resources. To do so, I created a "survey" and sent it to over 250 people to run on their computers. I received 22 responses, representing roughly an 8.5% response rate. The survey ran on each person's computer, taking observations roughly over a two day period. The survey is made up of a shell script that makes each observation and outputs the data to files, in addition to two python scripts which were used to anonymize the data and find the process and user owning the other end of UNIX domain sockets, pipes, and fifos. This survey gathered information about all open pipes, named pipes, UNIX domain sockets, TCP sockets, and UDP sockets. The tool used to find this information is called `lsof` which stands for list open files.

For each computer, there were eleven files collected. One file contained the anonymized output of the `ps` command, which describes all of the processes currently running on the machine. This data is used to connect helper processes with the application that they work for. I also collected the anonymized raw output of `lsof` after filtering for a specific type of local IPC. Therefore, there are five files containing this data, one each for anonymous pipes, named pipes, UNIX domain sockets, TCP sockets and UDP sockets. In addition, for each of these types of local IPC, there is another file that represents how many of each type a process has open at one time. For example, instead of six lines showing the different UNIX domain sockets that Spotify has open at one time, one observation in this file would have the process name, Spotify, and the number of local sockets, 6, without any of the other information outputted by `lsof`. All of these files were uploaded to the Middlebury College Computer Science Department's machine, basin.

All identifying information was anonymized so that no data can be attributed to any individual. To remove all username mentions, I used the `dscl` utility to find a list of all users on the computer. I removed some users from this list such as `root`, `nobody`, and `Guest` who did not need to be anonymized. Then, for each file that could possibly contain a username, I ran a Python script that replaced each occurence of a listed username surrounded by word boundaries with USERNAME. In doing this, I lost the ability to differentiate between different user accounts on a single computer, but it is unlikely that different human users were running applications since all computers were personal computers. I was, however, able to tell between a human user and either `root` or another automated account, such as `_mDNSResponder`.

I also wanted to be able to see if connections were being made between processes running under different users, specifically between a normal user and `root`. I made another Python script that found this information for UNIX domain sockets, pipes, and named pipes. First, for each individual end of a UNIX socket, pipe, or FIFO, it found the user owning that end and the device number corresponding to it. Then, it looked in the column representing the other end of the communication, and matched the process and user that was on the other end. This way, I could see which users and processes were communicating with each other.

4.2 Extracting Results

Once I had all of the data, I created another Python script to extract the results to inform my decision of what applications to look into. First, I looked through all of the processes that were running and manually created groups of processes that were part of the same applications. For some, like *Spotify* and *Spotify Helper*, it was easy to tell that both processes are part of the application *Spotify*. However, for others, like *Safari* and *com.apple.Webkit.Networking*, it took some additional research to know that these processes both were part of the *Safari* application.

Then, I created a table that contained the results for every application I manually recognized, or for each process if I did not assign it to a particular application. For each application, the data table contains the name of the application, the number of different computers that had this application running at least once, and the average number of named pipes, anonymous pipes, local and total TCP and UDP sockets, and UNIX domain sockets open at any time. It also contains all of the users that had at least one end of a communication channel with that application.

This data, particularly the average number of open FIFOs, local TCP and UDP sockets, and UNIX sockets, was what I used to decide what applications were the most important to examine. Another important aspect which is not reflected in this table was whether an application's UNIX socket was named. This was important because I cannot connect to an unnamed UNIX socket, created through the `socketpair` system call, without explicitly being given an endpoint. Therefore, if an application had UNIX sockets and at least one was named in the filesystem, then I was more likely to choose this application because it is more likely to have interesting results.

4.3 Fuzzing

As described in 3.4, fuzzing is the process of sending random or semi-random data to a communication endpoint in an attempt to crash the receiving process. I decided to use *radamsa* [3] for my fuzzing software. I chose *radamsa* for a variety of reasons. First, it is an easy-to-install fuzzer. Because of the short timeframe I have for this thesis, I needed a high-quality fuzzer that would not take long to tune or understand how it works. *radamsa* provides just that. All that is needed is to install the source code and compile it, and the resulting executable is all that is required to successfully fuzz.

The second reason that I chose *radamsa* is because it has no frills. All that *radamsa* does is take valid input and randomly modify it. There is fancy interface to use and more importantly, no parameters to tune. I send a valid example of a message as input and an altered string is returned. This message can then be encapsulated within the packaging required to send the data over the desired communication channel. This allows *radamsa* to easily be included in a script that generates semi-random input, sends it to the tested endpoint, and checks to make sure that the program is still running. That is exactly what I did.

4.4 Seeing Local IPC Communication

To focus in my fuzzing efforts and to make the fuzzing more effective, I needed to be able to see what messages open sockets and pipes were sending messages. For communication over the loopback interface, this was simple. I downloaded and installed *Wireshark*, an application that shows the raw bytes being sent through different network interfaces and identifies the different headers from each layer when it is able to understand a packet. This way, I was able to see every bit of every message that went through the loopback interface.

Named pipes and UNIX domain sockets did not have a similar, easy way to view all messages. Once I found the name of a named pipe in the filesystem, I could join the pipe as a reader. However, I would not see every message sent through the pipe if there were other readers. Each message is only received by a single reader. Therefore, if I was one of many readers, the single process that was given a message was randomly selected. When testing this, neither the user reading from the pipe, either root or my user, not the order in which a process became a reader for the pipe affected how many messages a process was given.

If the named pipe was not random, then I could delete the pipe from the filesystem and create a new one with the same name. Doing this, I was trying to see what messages were being sent from the clients to the server. This would give me messages that I could modify to fuzz the servers. Since most communication follows the request/response pattern, I received most of my sample messages by impersonating the server and capturing these genuine requests.

Communication over UNIX domain sockets is even more difficult to view than named pipes. For sockets that are created by `socketpair`, it is impossible to view their data without being explicitly given an endpoint. I could not get any of these endpoints in my testing, so these messages were completely hidden from my view. However, it is sometimes possible to view data sent through a named UNIX domain socket. Like Internet sockets, UNIX domain sockets know information about the other side of the communication, and can create separate communication channels. For example, for stream sockets, each connection that a listening socket accepts is separate from one another. The server is able to differentiate between the different clients, and messages sent to one connection are not sent to all others. Therefore, to receive messages from the clients, one must create the UNIX domain socket before the true application does. Receiving messages from the server is more difficult, because I had to connect to the server and send messages that elicit a response. Essentially, I needed to fuzz the server without any data to base the packets on and hope that one of the random packets get a response.

In the next chapter, I will go into more depth on how I found sample messages for different forms of local IPC for the applications that I studied. I will also discuss the fuzzing and other tests I ran to find how secure these communications are.

CHAPTER 5
RESULTS

This chapter discusses how I took the results of my survey and began to look at the different ways that applications use local IPC, as well as my work testing the security of this communication.

5.1 General Results

Based on the results of the survey that I received, I am able to make some general statements about the way that the three types of local IPC that I am studying–named pipes, internet sockets that use the loopback interface, and UNIX domain sockets–are used by applications. All of this data is reflected in Table C. I received twenty-two responses from the over 250 people that my survey was sent out to, roughly a 9% response rate. I was hoping to have more data, but was able to find results, even with the limited responses.

I collected data on the number of anonymous pipes that an application had open to be able to compare to the local IPC methods I studied. Anonymous pipes were used the most, with many applications having tens, or even hundreds of them open at a time. While these are used the most, I consider them to be a secure form of local IPC since, similar to `socketpair` sockets, the process creating one must explicitly give another process access.

The most notable result when looking through my data is how little named pipes are used by applications. Only three applications used named pipes at all, and all three were applcations created by Adobe. In the Man-in-the-Machine paper that prompted my topic [11], named pipes were one of the two most-common ways that the authors exploited local IPC. However, all of their named pipe exploits occurred only on Windows computers. This could imply that they also found few Mac applications that used named pipes or that applications running on Mac and Linux operating systems use UNIX domain sockets instead of named pipes. Because of the low number of applications that utilized named pipes, I did not study their security, so the rest of this chapter will almost exclusively be about internet sockets and UNIX domain sockets.

Internet sockets using TCP and UDP and connecting over the loopback interface were less commonly used that anonymous pipes. Only three applications had at least ten TCP or UDP sockets open and listening on the loopback interface at any time. However, one of those, mDNSResponder, averaged over sixty listening UDP sockets at a time. Other than this outlier, there was little difference between the number of TCP or UDP sockets open at a time. This could be due to the low chance that a UDP packet, even though it does not have guaranteed delivery, is dropped in transit, so many applications that may use TCP for networked communication opt for the lower overhead of UDP for local communication.

Finally, UNIX domain sockets are used on average more than named pipes or internet sockets. Twenty-one applications had at least ten UNIX sockets open at a time, with seven of these having at least thirty-five open on average. These statistics do not differentiate between named unnamed `socketpair` UNIX sockets, but from examining the detailed output of my survey, it looks like most UNIX domain sockets are created by the `socketpair` system call.

5.2 Applications

I decided to look at four different applications: Spotify, Microsoft Visual Studio Code (Code), launchd, and mDNSResponder. These four applications gave me broad coverage of many factors that I wanted to look into. Spotify and Code are user programs that use multiple processes to work together. On the other hand, launchd is a single process that is the first process created at boot time, giving it the process ID of 1. It is also owned by root, not a normal user. mDNSResponder is an application that uses two processes, one owned by root and the other owned by the _mdnsresponder user.

These applications also use different forms of local IPC and were open often in my survey. As stated above, very few applications on OS X use named pipes, so I did not study how applications used FIFOs since they were so rare. Spotify was being run on over 50% of surveyed machines, and on average, it had one TCP port and three UDP ports listening on the localhost address. It also averaged ten UNIX domain sockets open at any time. Code was run on relatively few machines, just under 10%, but had two listeing TCP sockets and fourteen UNIX domain sockets, many of them named in the filesystem. launchd and mDNSResponder were both running on 100% of computers. launchd had four local TCP sockets listening, two UDP sockets, and thirty-six UNIX domain sockets open on average. mDNSResponder had one local TCP socket, sixty-two local UDP sockets, and forty-four UNIX domain sockets. launchd and mDNSResponder also both had named UNIX sockets, not only `socketpair` sockets.

These applications also use local inter-process communication between different users. For example, Spotify has UNIX domain sockets between its processes, which are owned by a normal user. However, it also has sockets where the user at the other end is root or _mdnsresponder. These differences in endpoint privileges could be the basis for privilege escalation or execution hijacking as root user, and therefore are especially worth studying because of the increased risk.

Next, I will go into more depth on each application, describing its local IPC footprint as well as detailing the steps I took while investigating its security. This will include fuzzing TCP and UDP endpoints through the loopback interface and UNIX domain sockets.

5.3 Spotify

Spotify has a broad local IPC footprint. Each instance of the Spotify application runs two different processes, Spotify and Spotify Helper. Additionally, a third type of process also is used occasionally, SpotifyWebHelper, but this was only seen on two machines in the survey and I was unable to reproduce it while testing. When I was testing the Spotify application, it had two TCP sockets listening on any address on ports 57621 and 57819. It also had three UDP sockets listening on ports 1900, 62152, and 57621. Finally, it had eleven open UNIX domain socket endpoints. We will dive into each of these sections deeper.

5.3.1 UDP

Spotify has three UDP ports listening on any IP address when it runs. One is always listening on port 1900 and another is on a random port. Unfortunately, I was unable to find any communication over these ports.

However, the last UDP port is listening on port 57621. This port sends a regular message every thirty seconds. I believe that this is a “heartbeat” message sent to the local broadcast address for both the loopback interface and the Wifi radio interface of my computer. This message could be used as a way to find other instances of Spotify running nearby, or as a way for some software to see that this instance of Spotify is still running. The data of each message is always exactly forty-four bytes long, with the first eight bytes reading SpotUdp0. While the data is different for each instance of Spotify that I run, within an instance, the data is always the same, meaning once I start Spotify, the data sent every thirty seconds will always be exactly the same. Since I have many examples of these messages, I was able to effectively fuzz this endpoint.

I created a Python program that would send the two pieces of the data section to *radamsa* and use the result to build a new message. I then sent this new, random message to UDP port 57621 via localhost. I did this over 500,000 times. After each message, I checked to see whether Spotify was still running. None of these random strings caused a crash in Spotify.

Even though I was unable to find any messages to the other two UDP ports, I still fuzzed them using the same messages that I fuzzed port 57621 with. I fuzzed both of these ports over 500,000 times as well and never crashed Spotify. I then moved on to the TCP sockets to see if I would be able to find a bug while fuzzing these.

5.3.2 TCP

When the Spotify application runs, it establishes many TCP connections across the Internet to get the music and other data for the user. However, there are also two TPC sockets that listen on any interface. One is always on port 57621 while the other is random. Using Wireshark, I could not find any messages being sent on either of these ports for any instance of Spotify.

Again, like the last two UDP ports, I still fuzzed these endpoints, even without finding any messages being sent to these ports. When I fuzzed TCP port 57621 using the UDP heartbeat message, I was unable to cause a crash in Spotify. However, when I fuzzed the random TCP port, something else happened. Each time I created a connection, the socket would immediately be closed for me to write. When I investigated further, whenever I connected to this socket, I immediately received a message stating that the type is Tier1 and the version is 1.0. Then, the socket would be closed, all before I would be able to send a message. After doing some research, I believe that this message is stating the version of *node* being used by Spotify. Unfortunately, since the socket was being closed immediately, I was unable to fuzz it.

5.3.3 UNIX Domain Sockets

Finally, I investigated the UNIX domain sockets that Spotify opened. For each instance of Spotify created during testing, the Spotify application would have eleven UNIX domain socket endpoints open. These were split among one instance of the Spotify process and two instances of the Spotify Helper process. Of these eleven endpoints, eight were created using `socketpair` and shared between the various Spotify processes. There was one pair within the Spotify process, and another pair between each Spotify Helper process and the Spotify process. Finally, there was a pair between the two Spotify Helper processes. Since no other process was given any of these endpoints, there was no way to send any messages or attempt to fuzz these endpoints. However, there were also three UNIX domain sockets that were connected to named UNIX sockets. One was connected to a socket owned by both launchd and syslogd. The security of this communication will be discussed in Section 5.5.3. The other two were both connected to a socket owned by mDNSResponder. This will be examined in Section 5.6.3.

5.3.4 Spotify Conclusion

Based on my research into Spotify, I believe that this application is fairly secure against local input-based vulnerabilities. I was unable to crash Spotify using any of the open UDP or TCP sockets. Additionally, its use of UNIX domain sockets is very secure since almost all of them are created using `socketpair` and therefore cannot be directly given input by an outside process.

5.4 VS Code

VS Code only uses two of the forms of local IPC that I studied: local TCP connections and UNIX domian sockets. However, the biggest reason that I chose to look at VS

Code is because it has so many named UNIX sockets. Additionally, some of these sockets have the same name for each instance of VS Code while others have random suffixes each time. This gives an interesting contrast to study. VS Code is made up of the processes Code Helper and Electron. There is always only on running process of Electron, but depending on the number of windows and tabs open, there may be many running versions of Code Helper.

5.4.1 TCP

VS Code averaged two listening TCP ports, according to my survey data. The ports chosen were always random, so I needed to find the port each time I started a fuzzing session. When I originally fuzzed these ports with random data, I was always returned the message "WebSockets request was expected." After doing research, I found that this meant that it was looking for an HTTP request. Therefore, I created an HTTP GET request to get the repository holding the research for this thesis from GitHub and used this as my genuine communication example.

When I used this message to fuzz the Code endpoint, the program would end between fifty and one hundred messages into fuzzing because the port had been closed. Whenever I went back to my open instance of VS Code, my VIM extension no longer worked. VS Code was no longer using VIM keybindings nor was it using the normal keyboard keys and shortcuts. This experiment was repeated three times, and each time the VIM extension crashed. I had crashed this extension. This finding represents a bug in the way that VS Code parses input to its random TCP ports since it does not correctly handle the input I was sending. This bug could also constitute a vulnerability, which, if exploited, could allow an attacker to either hijack execution or disclose personal information. Therefore, this is a significant finding.

5.4.2 UNIX Domain Sockets

In addition to its TCP sockets, VS Code also has many UNIX domain sockets. Four of these sockets are named. One is named based on the version number and the word ‘main’, hereafter known as the main socket. One is named off of the version number and the word ‘shared’, known as the shared socket. The last two are named using either the word ‘git’ or ‘ipc’ and followed by a random suffix; these are the git and ipc sockets, respectively. These random socket names were likely created with the `mktemp` system call or utility to create a name that is guaranteed to be unique in the filesystem.

When fuzzing the main socket, I used a six-byte message as my seed that was sent when the application was being closed. However, after fuzzing over 500,000 times with variations of this message, VS Code did not crash or function incorrectly.

However, since the name of the main socket does not change, I deleted the socket from the filesystem and created it again myself. Then, I tried to open VS Code. I received a message saying that another instance of VS Code was running. I had created a Denial-of-Service attack by opening the main socket myself. VS Code must have checked the return value of the bind call, seen that this name in the filesystem was already taken, and decided that an instance was already running. Since I had created this socket, no other version of VS Code would be able to run without first deleting the socket from the file system and killing any process that had it open or restarting the entire machine.

The next socket that I looked at was the shared socket. By impersonating this socket, I found a forty-five byte message that was sent from a client when VS Code started. Using this message to fuzz the shared socket, I was unable to cause a crash after 500,000 iterations.

I then investigated the git socket. Since it has a random suffix, I could not create this socket before I had a running instance of VS Code. However, I did delete the socket and recreate it myself, but did not find any sample messages. When I attempted to fuzz this socket, I received a BrokenPipeError in python after two messages were successfully sent. A BrokenPipeError in python means that the remote communication endpoint has closed the socket for reading, but the local endpoint is still sending data. I hypothesize, since this error always occurred after two successful sends, that this could be a security measure put in place by VS Code. This socket may never expect to see more than two messages from the same client, so if it ever sees a third message, it closes that connection for reading because it believes something nefarious is happening. It is also possible that this could be an error in the way that the socket is being used, but since it always occurred after exactly two successful messages, I believe that it is for security purposes.

The last named socket is the ipc socket. Again, this socket has a random suffix so I cannot create it before I started VS Code. Once I started VS Code, I would see in the output of `lsof` that this socket was open and connected to another UNIX domain socket owned by another instance of the Code Helper process. However, when I tried to delete this socket from the file system to create my own version, I found that this socket did not exist. If I tried to connect to the ipc socket as a client, I would receive an error stating that the file did not exist.

This means that VS Code created this socket and created the connections it needed while the application was starting, then immediately called `unlink` on the socket. This would remove the link count of the socket, decrementing it to zero. Since the link count was zero, it would no longer show up in the filesystem, which explains my findings. However, since this socket was still being used, it would not actualyl be garbage collected by the system until all processes referencing the file ended. Therefore, the two separate Code Helper processes were able to connect and still communicate while being sure that no other process could snoop on the communication.

I do not know why VS Code uses this system for the ipc socket, especially when it has so many `socketpair` sockets open at the same time, even between the two processes connected by the ipc socket. However, the usage of the ipc socket is, in practice, the same as a `socketpair` socket since no other process can join the conversation.

5.4.3 VS Code Conclusion

My research has shown that VS Code is not immune against input-based vulnerabilities over local IPC. Through my fuzzing, I crashed a VIM extension that was open in VS Code. Additionally, I caused a Denial-of-Service attack by opening the main UNIX socket before opening the VS Code application. While VS Code does use some security features, like closing the connection if the git socket receives more than two messages and immediately `unlink`ing the ipc socket, it still is vulnerable and must be fixed, lest it be exploited.

5.5 launchd

I chose to text launchd because it is the process that helps boot the machine, as well as starting many services, or daemons. It is responsible for creating and maintaining many necessary services, such as networking, logging, and showing images on the screen. launchd uses local TCP and UDP sockets, as well as many named UNIX domain sockets.

5.5.1 TCP

My survey results showed that launchd averaged four open TCP sockets at a time. However, no matter when I tried to find these sockets on my own computer, I was unable to find an open TCP socket that launchd was listening to. Therefore, since I could not recreate this result, I was unable to fuzz any TCP sockets for launchd.

5.5.2 UDP

launchd averaged two open UDP sockets, one listening on port 137 and the other on port 138, both listening on any interface. These ports are the netbios-ns and netbios-dgm ports. I used *Wireshark* to find sample messages being sent to these ports, and found two examples of mDNS responses. To fuzz these sockets, my program would randomly choose one message, send it to *radamsa* to modify, and then randomly choose a port to send it to. In this way, both messages were randomly being sent to both ports, maximizing the possiblity of a crash. Even with this process, fuzzing both of these ports did not cause launchd to crash.

5.5.3 UNIX Domain Sockets

Unlike every other process that I looked at, a majority of launchd's UNIX domain sockets were not `socketpair` sockets. In face, launchd had no `socketpair` endpoints; it only had named UNIX domain sockets. While the survey results show that launchd had on average thirty-six open UNIX domain sockets, each socket was opened twice, so there were only eighteen different sockets open. I attempted to fuzz six of these, representing one-third of the open sockets. I was unable to find sample messages for any of these UNIX domain sockets, so I used the same sample messages from fuzzing the launchd UDP sockets as the seeds for my UNIX domain socket messages.

The first socket that I fuzzed was the socket that almost every running application connected to that did logging. Spotify, VS Code, and mDNSResponder all had a connection to this socket. This socket was a datagram UNIX socket, and the only one that I found in my research. I fuzzed this socket over 500,000 times and was not able to crash launchd. I had the same result with two stream sockets that worked with remote procedure calls and a VPN.

The other three UNIX sockets that I attempted to fuzz were part of the system keychain, a port-mapping software, and the software that multiplexes access to the USB port. While fuzzing these sockets, a BrokenPipeError would be thrown often, and quite regularly. The system keychain socket would throw the error after one or two messages was sent successfully, then to every message for a short time period after. Similarly, the port-mapping socket would through the error after about forty messages, and then to every message from every connection for a short time. This seems to show that the process owning the socket sees that too many messages are being sent, and could be malicious. It then closes the socket for reading for a short amount of time to hopefully discourage the malicious user from continuing.

The USB socket also would throw a BrokenPipeError after the first message it received from every single connection. Like with the git UNIX domain socket from VS Code in Section 5.4.2, this could be a security feature. Like with VS Code, it is difficult to tell the exact motivation behind this function, or even if it is intended. However, based on my research, it is plausible that this error is meant to decrease the likelihood of an input-based attack.

5.5.4 launchd Conclusion

From my research, launchd seems to be secure against many local IPC-based input attacks. I could not recreate its TCP sockets and fuzzing its UDP sockets never caused a crash. While it has many named UNIX domain sockets, it appears that many of these implement some sort of security plan to diminish the possibility of input-based attacks. The sockets that do not seem to have this plan did not cause a crash of launchd while being fuzzed, so they are secure as far as I have tested.

5.6 mDNSResponder

The final process that I tested is mDNSResponder, a process responsible for different parts of networking, including DNS, AirDrop, and finding nearby printers. mDNSResponder had the largest local IPC footprint of the four applications that I studied, averaging sixty-two listening UDP sockets and forty-four UNIX domain sockets. This set up a large testing ground for my research.

5.6.1 TCP

According to my survey, mDNSResponder had, on average, one TCP socket listening on any interface. However, this socket was always in the 'CLOSED' state, which means that it has not started to listen yet, or had a connection and closed it and is not yet listening for any more. Therefore, since this socket was not listening for incoming connections, I could not fuzz the single, local mDNSResponder TCP socket.

5.6.2 UDP

mDNSResponder always has two sockets listening on the mdns port, port number 5353. This can be done by setting the SO_REUSEPORT on all of the sockets before binding, and can help load-balance across multiple sockets. I found two sample messages send to port 5353 by using *Wireshark*, and used these as my seed messages for fuzzing. Each iteration, one of the two messages was randomly selected and then randomly modified, then sent to the port. I was unable to cause a crash of mDNSResponder or a slowdown in Internet speed from this fuzzing.

5.6.3 UNIX Domain Sockets

As mentioned above, mDNSResponder has over forty UNIX domain sockets open at any one time. There are four endpoints open within the process, representing four of the total UNIX socket endpoints. The other forty are all connected to the same named UNIX domain socket. I fuzzed this named socket using a thirty-two-byte message that I received when I created my own version of this socket. Even after 500,000 iterations, Internet traffic was no slower and mDNSResponder had not crashed.

However, when I created my own instance of the named UNIX socket after deleting the real version, most Internet traffic from my computer failed. All webpages failed to load, and I could no longer get music from Spotify. This was essentially a Denial-of-Service attack, where I denied access to most of the rest of the Internet. Specifically, any service that required DNS would not work. The only way to bring back Internet access was to restart the entire machine and let mDNSResponder create the real socket again.

5.6.4 mDNSResponder Conclusion

Based on all of my research, it seems that mDNSResponder is secure against local input-based vulnerabilities. However, if its named UNIX domain socket is deleted, the machine loses most access to the rest of the outside world. While this attack is not an input-based attack, it uses local IPC channels as the attack vector to stop Internet access.

CHAPTER 6

FUTURE WORK

WHAT ELSE COULD YOU RESEARCH ABOUT LOCAL IPC

ONLY LOOKED AT BUGS THAT CRASHED THE SOFTWARE, OTHER WAYS TO MEASURE UNEXPECTED EXECUTION

CHAPTER 7

DISCUSSION

We HAVE FOUND X BUGS/VULNERABILITIES IN THESE APPLICATIONS

POSSIBLE FIXES FOR THESE IN THE FUTURE

APPENDIX A

CHAPTER 1 OF APPENDIX

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

APPENDIX B

CODE SAMPLES

This appendix contains samples of code to use the three forms of local IPC that I studied in my thesis. There are examples of reading from and writing to a named pipe, communicating through TCP and UDP over the loopback interface, and using stream- and datagram-based UNIX domain sockets.

B.1 Writing to a Named Pipe

```
/*
 * pipeWriter.c
 */

// C program to open and write to a FIFO

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

// path to pipe
#define PIPE_PATH "./my_pipe"
#define MAX_BYTES 100
```

```
int main()
{
        char buf[MAX_BYTES];
        // Open FIFO as writer
        int fd = open(PIPE_PATH, O_WRONLY | O_CREAT);
        if (fd < 0)
        {
                perror("open");
                exit(1);
        }
        while (1)
        {
                // get input from user
                char *result = fgets(buf, MAX_BYTES, stdin);
                if (result == NULL)
                {
                        perror("fgets");
                        exit(1);
                }

                // write input into the pipe
                int writeRes = write(fd, buf, strlen(buf)+1);
                if (writeRes < 0)
                {
                        perror("write");
                        exit(1);
                }
        }
        fd = close(fd);
        if (fd < 0)
        {
                perror("close");
                exit(1);
        }
        return 0;
}
```

B.2 Reading from a Named Pipe

```
/*
 * pipeReader.c
 */

// C program to open and read from a FIFO

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
```

```
// path to pipe
#define PIPE_PATH "./my_pipe"
#define MAX_BYTES 100

int main()
{
        char buf[MAX_BYTES];
        // Open FIFO as writer
        int fd = open(PIPE_PATH, O_RDONLY | O_CREAT);
        if (fd < 0)
        {
                perror("open");
                exit(1);
        }
        while (1)
        {
                int readRes = read(fd, buf, MAX_BYTES);
                if (readRes < 0)
                {
                        perror("read");
                        exit(1);
                }
                else if (readRes == 0)
                {
                        continue;
                }
                printf("Received: %s\n", buf);
        }
        fd = close(fd);
        if (fd < 0)
        {
                perror("close");
                exit(1);
        }
        return 0;
}
```

B.3 Loopback Interface TCP Server

```
/*
 * streamLocalhostServer.c
 */

// This program creates a stream Internet socket over localhost

#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <string.h>
```

```
#include <unistd.h>

#define PORT "50000"
#define IP_ADDR "localhost"
#define BACKLOG 10

int main(int argc, char *argv[])
{
    // create the socket
    int sockFd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockFd < 0)
    {
        perror("socket");
        exit(1);
    }

    // Bind it to a port
    struct addrinfo hints;
    struct addrinfo *res;
    int rc;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if((rc = getaddrinfo(NULL, PORT, &hints, &res)) != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rc));
        exit(1);
    }

    int retVal = bind(sockFd, res->ai_addr, res->ai_addrlen);
    if (retVal < 0)
    {
        perror("bind");
        exit(1);
    }

    // Start listening
    retVal = listen(sockFd, BACKLOG);
    if (retVal < 0)
    {
        perror("listen");
        exit(1);
    }

    int readResult = 10;
    char buf[100];
    while (1)
    {
        // accept each connection and read the first message sent the
        int clientFd = accept(sockFd, NULL, NULL);
        if (clientFd < 0)
        {
            perror("accept");
            continue;
        }
        readResult = recv(clientFd, buf, sizeof(buf), 0);
        if (readResult < 0)
        {
            perror("recv");
            exit(1);
        }
        printf("Received: %s", buf);
        memset(buf, 0, sizeof(buf));
        clientFd = close(clientFd);
        if (clientFd < 0)
        {
            perror("close");
            exit(1);
        }
    }
}
```

B.4 Loopback Interface TCP Client

```
/*
 * streamLocalhostClient.c
 */

// This program connects to a stream Internet socket over localhost
```

```
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>

#define PORT "50000"
#define IP_ADDR "localhost"

int main(int argc, char *argv[])
{
    // create the socket
    int sockFd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockFd < 0)
    {
        perror("socket");
        exit(1);
    }

    struct addrinfo hints;
    struct addrinfo *res;
```

```
int rc;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
if((rc = getaddrinfo(IP_ADDR, PORT, &hints, &res)) != 0)
{
    printf("getaddrinfo failed: %s\n", gai_strerror(rc));
    exit(1);
}

// Connect to server
if(connect(sockFd, res->ai_addr, res->ai_addrlen) < 0)
{
    perror("connect");
    exit(1);
}

int bytesRead = 10;
char buf[100];
// read in from stdin and send
bytesRead = read(0, buf, 100);
if (bytesRead < 0)
{
    perror("read");
    exit(1);
}
```

```
    int retVal = send(sockFd, buf, bytesRead, 0);
    if (retVal < 0)
    {
        perror("send");
        exit(1);
    }
    int ret = close(sockFd);
    if (ret < 0)
    {
        perror("close");
        exit(1);
    }

}
```

B.5 Loopback Interface UDP Server

```
/*
 * datagramLocalhostServer.c
 */

// This program creates a datagram Internet socket over localhost

#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
```

```
#include <netdb.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>

#define PORT "50000"
#define IP_ADDR "localhost"

int main(int argc, char *argv[])
{
    // create the socket
    int sockFd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockFd < 0)
    {
        perror("socket");
        exit(1);
    }

    // Bind it to a port
    struct addrinfo hints;
    struct addrinfo *res;
    int rc;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE;
```

```
if((rc = getaddrinfo(NULL, PORT, &hints, &res)) != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rc));
    exit(1);
}

int retVal = bind(sockFd, res->ai_addr, res->ai_addrlen);
if (retVal < 0)
{
    perror("bind");
    exit(1);
}

int readResult = 10;
char buf[100];
while (1)
{
    memset(buf, 0, sizeof(buf));
    // accept each connection and read the first message sent the
    readResult = recvfrom(sockFd, buf, sizeof(buf), 0, NULL, 0);
    if (readResult < 0)
    {
        perror("recvfrom");
        exit(1);
    }
    printf("Received: %s", buf);
}
```

```
}
```

B.6 Loopback Interface UDP Client

```
/*
 * datagramLocalhostClient.c
 */

// This program connects to a datagram Internet socket over localhost

#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>

#define PORT "50000"
#define IP_ADDR "127.0.0.1"

int main(int argc, char *argv[])
{
    // create the socket
    int sockFd = socket(AF_INET, SOCK_DGRAM, 0);
```

```
if (sockFd < 0)
{
    perror("socket");
    exit(1);
}

struct sockaddr_in servaddr;
memset(&servaddr, 0, sizeof(servaddr));

// Filling server information
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(atoi(PORT));
inet_pton(AF_INET, IP_ADDR, &servaddr.sin_addr);

struct sockaddr *ptr = (struct sockaddr *) &servaddr;
int size = sizeof(struct sockaddr);

int bytesRead = 10;
char buf[100];
// read in from stdin and send
bytesRead = read(0, buf, 100);
if (bytesRead < 0)
{
    perror("read");
    exit(1);
}
```

```
    int retVal = sendto(sockFd, buf, bytesRead, 0, ptr, size);
    if (retVal < 0)
    {
        perror("sendto");
        exit(1);
    }
    int ret = close(sockFd);
    if (ret < 0)
    {
        perror("close");
        exit(1);
    }

}
```

B.7 UNIX Domain Socket Stream Server

```
/*
 * streamUnixServer.c
 */

// This program makes a stream UNIX domain socket with a name

#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/un.h>
```

```
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *socketPath = "named";

    // create the socket
    int sockFd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sockFd < 0)
    {
        perror("socket");
        exit(1);
    }

    struct sockaddr_un addr;
    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, socketPath, sizeof(addr.sun_path)-1);
    unlink(socketPath);

    // remove the socket if it exists
    int unlinkVal = unlink(socketPath);

    // bind the socket to the pathname
    int retVal = bind(sockFd, (struct sockaddr*) &addr, sizeof(addr))
```

```
if (retVal < 0)
{
    perror("bind");
    exit(1);
}

// listen for incoming connections
retVal = listen(sockFd, 10);
if (retVal < 0)
{
    perror("listen");
    exit(1);
}

int readResult = 10;
char buf[100];
while (1)
{
    // accept each connection and read the first message sent the
    int clientFd = accept(sockFd, NULL, NULL);
    if (clientFd < 0)
    {
        perror("accept");
        continue;
    }
    readResult = recv(clientFd, buf, sizeof(buf), 0);
```

```
        if (readResult < 0)
        {
            perror("recv");
            exit(1);
        }
        printf("Received: %s", buf);
        memset(buf, 0, sizeof(buf));
        clientFd = close(clientFd);
        if (clientFd < 0)
        {
            perror("close");
            exit(1);
        }
    }
}
```

B.8 UNIX Domain Socket Stream Client

```
/*
 * streamUnixClient.c
 */

// This program connects to a stream UNIX domain socket

#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/un.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *socketPath = "named";

    // create the socket
    int sockFd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sockFd < 0)
    {
        perror("socket");
        exit(1);
    }

    struct sockaddr_un addr;
    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, socketPath, sizeof(addr.sun_path)-1);

    // connect the socket to the listening socket
    int retVal = connect(sockFd, (struct sockaddr*) &addr, sizeof(add
    if (retVal < 0)
    {
        perror("connect");
        exit(1);
    }

    int bytesRead = 10;
    char buf[100];
    // read in from stdin and send
    bytesRead = read(0, buf, 100);
    if (bytesRead < 0)
    {
        perror("read");
        exit(1);
    }
    retVal = send(sockFd, buf, bytesRead, 0);
    if (retVal < 0)
    {
        perror("send");
        exit(1);
    }
    int ret = close(sockFd);
    if (ret < 0)
    {
        perror("close");
        exit(1);
    }

}
```

B.9 UNIX Domain Socket Datagram Server

```
/*
 * datagramUnixServer.c
 */

// This program makes a datagram UNIX domain socket with a name

#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/un.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *socketPath = "named";

    // create the socket
    int sockFd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sockFd < 0)
    {
        perror("socket");
        exit(1);
    }
```

```
struct sockaddr_un addr;
memset(&addr, 0, sizeof(addr));
addr.sun_family = AF_UNIX;
strncpy(addr.sun_path, socketPath, sizeof(addr.sun_path)-1);
unlink(socketPath);

// remove the socket if it exists
int unlinkVal = unlink(socketPath);

// bind the socket to the pathname
int retVal = bind(sockFd, (struct sockaddr*) &addr, sizeof(addr))
if (retVal < 0)
{
    perror(”bind”);
    exit(1);
}

int readResult = 0;
char buf[100];
while (1)
{
    memset(buf, 0, sizeof(buf));
    // read and print every message sent
    readResult = recvfrom(sockFd, buf, sizeof(buf), 0, NULL, 0);
    if (readResult < 0)
    {
```

```
                perror("recvfrom");
                exit(1);
            }
            printf("Received: %s", buf);
        }
    }
```

B.10 UNIX Domain Socket Datagram Client

```
/*
 * datagramUnixClient.c
 */

// This program connects to a datagram UNIX domain socket

#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/un.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *socketPath = "named";

    // create the socket
```

```
int sockFd = socket(AF_UNIX, SOCK_DGRAM, 0);
if (sockFd < 0)
{
    perror("socket");
    exit(1);
}

// set up the destination socket
struct sockaddr_un addr;
memset(&addr, 0, sizeof(addr));
addr.sun_family = AF_UNIX;
strcpy(addr.sun_path, socketPath);

struct sockaddr *ptr = (struct sockaddr *) &addr;
int len = sizeof(addr);

int bytesRead = 10;
char buf[100];
int retVal;
while (bytesRead > 0)
{
    // read in from stdin and send
    bytesRead = read(0, buf, 100);
    if (bytesRead < 0)
    {
        perror("read");
```

```
            exit(1);
        }
        retVal = sendto(sockFd, buf, bytesRead, 0, ptr, len);
        if (retVal < 0)
        {
            perror("sendto");
            exit(1);
        }
    }
    int ret = close(sockFd);
    if (ret < 0)
    {
        perror("close");
        exit(1);
    }

}
```

B.11 `socketpair` UNIX Domain Sockets

```
/*
 * socketpair.c
 */

// This program creates two UNIX domain sockets using socketpair.
The process
// forks and the child reads from stdin and sends it to the parent, w
// the data out.

#include <sys/socket.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int sockets[2];

    // make the two sockets
    int retVal = socketpair(AF_LOCAL, SOCK_DGRAM, 0, sockets);
    if (retVal < 0)
    {
        perror("socketpair");
        exit(1);
    }

    // fork
    int childPid = fork();
    if (childPid < 0)
    {
        perror("fork"); // fork error
        exit(1);
```

```
}
else if (childPid == 0)
{
    // in the child, will close 0th socket fd
    retVal = close(sockets[0]);
    if (retVal < 0)
    {
        perror("close");
        exit(1);
    }
    // read from stdin and write to socket
    char buf[100];
    while (1)
    {
        retVal = read(0, buf, 100);
        if (retVal < 0)
        {
            perror("read");
            exit(1);
        }
        retVal = send(sockets[1], buf, retVal, 0);
        if (retVal < 0)
        {
            perror("send");
            exit(1);
        }
        }
    }
    else
    {
        // in the parent, will close 1st socket fd
        retVal = close(sockets[1]);
        if (retVal < 0)
        {
            perror("close");
            exit(1);
        }
        // read from socket and print out
        char buf[100];
        while (1)
        {
            retVal = recv(sockets[0], buf, 100, 0);
            if (retVal < 0)
            {
                perror("recv");
                exit(1);
            }
            printf("Received: %s", buf);
            memset(buf, 0, sizeof(buf));
        }
    }
```

```
}
```

APPENDIX C

COMPLETE DATA TABLE

Below is the complete data table that I assembled from the results of my survey. It shows the application name, the number of machines it was running on, and the local IPC footprint of that application. All of the columns below represent averages found from my survey.

APP NAME	MACHINES	FIFOS	PIPES	LOCAL TCP(TOTAL)	LOCAL UDP(TOTAL)	UNIX
accountsd	8	0	0	0 (0)	0 (0)	1
Acrobat Updater	2	0	3	0 (0)	0 (0)	2
ADDRESS BOOK	4	0	0	0 (2)	0 (0)	2
Adobe CEF Helper	2	0	4	0 (0)	0 (0)	5
Adobe Installer	1	2	0	0 (0)	0 (0)	0
AdobeCRDaemon	1	0	0	0 (0)	0 (0)	1
AdobeIPCBroker	2	0	0	0 (0)	0 (0)	10
AdobeReader	2	0	0	0 (0)	0 (0)	7
AdobeUpdateDaemon	2	2	0	0 (0)	0 (0)	0
AIRPLAY	22	0	0	0 (1)	2 (2)	5
airportd	22	0	0	0 (0)	10 (10)	3
akd	18	0	0	0 (0)	0 (0)	1
APPLE PHOTOS	1	0	0	0 (0)	0 (0)	4
APPLE PREFERENCES	1	0	0	0 (0)	3 (3)	5
appleeventsd	1	0	0	0 (0)	0 (0)	1
AppleIDAuthAgent	1	0	0	0 (0)	0 (0)	2
AppleMobileDeviceHelper	8	0	2	0 (0)	0 (0)	3
AppleSpell	1	0	226	0 (0)	0 (0)	1
AppStats	1	0	2	0 (0)	0 (0)	2
appstoreagent	14	0	0	0 (0)	0 (0)	1
AppTech	2	0	3	0 (0)	0 (0)	2
apsd	22	0	0	0 (2)	0 (0)	4
ARDAgent	1	0	0	1 (1)	2 (2)	1

askpermissiond	1	0	0	0 (0)	0 (0)	1
AssetCacheLocatorService	3	0	0	0 (0)	0 (0)	2
assistantd	13	0	0	0 (1)	1 (1)	0
ath	3	0	2	0 (0)	0 (0)	5
ATOM	5	0	36	0 (3)	0 (0)	10
authd	1	0	0	0 (0)	0 (0)	1
AVAST	1	0	8	2 (56)	0 (0)	84
avconferenced	9	0	0	1 (1)	4 (4)	1
awdd	1	0	0	0 (0)	0 (0)	1
backgroundtaskmanagementagent	19	0	0	0 (0)	0 (0)	1
Backup and Sync	1	0	7	0 (15)	0 (0)	2
bash	22	0	1	0 (0)	0 (0)	1
biometrickitd	2	0	0	0 (1)	0 (0)	1
bird	5	0	0	0 (0)	0 (0)	1
BlueJeans	1	0	6	1 (1)	0 (0)	0
BLUETOOTH	19	0	0	0 (0)	3 (3)	5
Calculator	1	0	0	0 (0)	0 (0)	1
CALENDAR	13	0	0	0 (2)	0 (0)	2
CALLHISTORY	1	0	0	0 (0)	0 (0)	2
callservicesd	2	0	0	0 (0)	0 (0)	1
CalNCService	1	0	0	0 (0)	0 (0)	1
captiveagent	15	0	0	0 (0)	0 (0)	2
CARBONITE	1	0	34	2 (5)	0 (0)	4
cfbackd	1	0	0	0 (0)	0 (0)	5
cfprefsd	5	0	0	0 (0)	0 (0)	1
CHROME	16	0	50	1 (26)	1 (8)	38
CiscoVideoGuardMonitor	1	0	8	2 (2)	0 (0)	0
cloudd	19	0	0	0 (2)	0 (0)	1
cloudfamilyrestrictionsd	1	0	0	0 (0)	0 (0)	1
cloudpaird	1	0	0	0 (0)	0 (0)	1
cloudphotosd	1	0	0	0 (0)	0 (0)	3
CMFSyncAgent	1	0	0	0 (0)	0 (0)	1

CODE	2	0	52	2 (10)	0 (0)	22
com.apple.AmbientDisplayAgent	4	0	0	0 (0)	0 (0)	1
com.apple.appkit.xpc.openAndSav	20	0	0	0 (0)	0 (0)	1
com.apple.BKAgentService	4	0	0	0 (0)	0 (0)	1
com.apple.CommerceKit.Transacti	3	0	0	0 (3)	0 (0)	2
com.apple.dock.extra	1	0	0	0 (0)	0 (0)	1
com.apple.geod	7	0	0	0 (2)	0 (0)	3
com.apple.hiservices-xpcservice	1	0	0	0 (0)	0 (0)	1
com.apple.iCloudHelper	2	0	0	0 (1)	0 (0)	1
com.apple.MediaLibraryService	3	0	0	0 (0)	0 (0)	1
com.apple.PerformanceAnalysis.a	3	0	0	0 (0)	0 (0)	1
com.apple.security.pboxd	17	0	0	0 (0)	0 (0)	1
com.apple.speech.speechsynthesi	1	0	0	0 (0)	0 (0)	1
com.apple.tonelibraryd	1	0	0	0 (0)	0 (0)	1
com.symantec.SymLUHelper	2	0	0	0 (0)	0 (0)	2
CommCenter	20	0	0	0 (0)	0 (0)	1
commerce	10	0	0	0 (2)	0 (0)	1
configd	22	0	0	0 (0)	4 (4)	3
Contacts	1	0	0	0 (0)	0 (0)	1
ContainerMetadataExtractor	10	0	0	0 (0)	0 (0)	1
ControlStrip	1	0	0	0 (0)	0 (0)	1
CORE SERVICES	2	0	0	0 (0)	0 (0)	2
coreaudiod	1	0	0	0 (0)	0 (0)	1
coreduetd	1	0	0	0 (0)	0 (0)	1
corespeechd	15	0	0	0 (0)	0 (0)	1
CrashReporterSupportHelper	17	0	0	0 (0)	0 (0)	1
CREATIVE CLOUD	2	2	19	2 (13)	0 (0)	22
cupsd	11	0	4	2 (2)	0 (0)	2
DashboardClient	2	0	0	0 (0)	0 (0)	1
DashlanePluginMASService	1	0	4	1 (1)	0 (0)	0
DataDetectorsDynamicData	1	0	0	0 (0)	0 (0)	1
dbfseventsd	6	0	9	0 (0)	0 (0)	2

digest-service	1	0	0	0 (0)	0 (0)	1
diskarbitrationd	1	0	0	0 (0)	0 (0)	1
displaypolicyd	19	0	0	0 (0)	0 (0)	1
distnoted	1	0	0	0 (0)	0 (0)	1
Dock	22	0	0	0 (0)	0 (0)	1
DocumentPopoverViewService	3	0	0	0 (0)	0 (0)	1
DrCleaner	1	0	0	0 (0)	1 (1)	2
DROPBOX	6	0	25	0 (11)	1 (1)	68
dsAccessService	4	0	6	0 (0)	0 (0)	0
eapolclient	21	0	0	0 (0)	1 (1)	2
EEventManager	1	0	0	1 (1)	1 (1)	0
EmojiFunctionRowIM_Extension	1	0	0	0 (0)	0 (0)	1
eoshostd	2	0	0	0 (1)	0 (0)	0
EPIC GAMES/FORTNITE	1	0	50	0 (3)	4 (4)	6
EpsonLowInkReminderAgent	1	0	0	0 (0)	0 (0)	1
EXCEL	9	0	0	0 (2)	0 (0)	3
EXPRESS VPN	2	0	2	1 (1)	1 (2)	1
FaceTime	1	0	0	0 (0)	0 (0)	1
familycircled	15	0	0	0 (0)	0 (0)	1
filecoordinationd	8	0	0	0 (0)	0 (0)	1
Finder	22	0	0	0 (0)	1 (1)	1
findmydeviced	20	0	0	0 (0)	0 (0)	1
FIREFOX	1	0	9	0 (19)	0 (0)	54
Flux	2	0	0	0 (0)	0 (0)	1
fmfd	4	0	0	0 (1)	0 (0)	3
fontd	1	0	0	0 (0)	0 (0)	1
FrontendAgent	2	0	0	0 (0)	0 (0)	1
fud	4	0	0	0 (0)	0 (0)	1
gamed	4	0	0	0 (0)	0 (0)	1
hidd	2	0	0	0 (0)	0 (0)	1
homed	12	0	0	0 (0)	0 (0)	3
HP Device Monitor	3	0	0	3 (3)	0 (0)	14

icdd	22	0	0	0 (0)	0 (0)	10
ICON SERVICES	1	0	0	0 (0)	0 (0)	2
identityservicesd	22	0	0	0 (4)	1 (1)	8
IDSKeychainSyncingProxy	1	0	0	0 (0)	0 (0)	1
Image Capture Extension	1	0	0	0 (0)	0 (0)	11
imagent	3	0	0	0 (0)	0 (0)	1
IMDPersistenceAgent	1	0	0	0 (0)	0 (0)	1
IMLoggingAgent	1	0	0	0 (0)	0 (0)	1
IMRemoteURLConnectionAgent	7	0	0	0 (3)	0 (0)	2
IMTransferAgent	2	0	0	0 (6)	0 (0)	0
ITUNES	11	0	2	0 (4)	0 (0)	10
java	2	0	0	1 (1)	0 (0)	5
Java Updater	1	0	0	0 (0)	0 (0)	2
JavaAppLauncher	2	0	0	0 (1)	0 (0)	3
kdc	2	0	0	2 (2)	2 (2)	1
KernelEventAgent	22	0	0	0 (0)	0 (0)	2
kextfind	1	0	1	0 (0)	0 (0)	0
keyboardservicesd	2	0	0	0 (0)	0 (0)	1
knowledge-agent	3	0	0	0 (0)	0 (0)	1
launchd	22	0	0	4 (4)	2 (2)	36
launchservicesd	1	0	0	0 (0)	0 (0)	1
locationd	22	0	4	0 (0)	1 (1)	3
LocationMenu	11	0	0	0 (0)	0 (0)	1
logd	19	0	0	0 (0)	0 (0)	1
loginwindow	22	0	0	0 (0)	1 (1)	1
LookupViewService	3	0	0	0 (4)	0 (0)	2
lsd	1	0	0	0 (0)	0 (0)	1
lsof	22	0	3	0 (0)	0 (0)	0
Magnet	1	0	0	0 (0)	0 (0)	1
MAIL	8	0	0	0 (4)	0 (0)	2
mapspushd	3	0	0	0 (0)	0 (0)	2
mDNSResponder	22	0	0	1 (3)	62 (62)	44

MDS	20	0	0	0 (0)	0 (0)	2
mdwrite	1	0	0	0 (0)	0 (0)	1
mediaremoted	10	0	0	0 (0)	0 (0)	1
Messages	16	0	0	0 (0)	0 (0)	1
Microsoft AutoUpdate	1	0	0	0 (0)	0 (0)	1
Microsoft Outlook	1	0	0	0 (0)	0 (0)	4
MirrorDisplays	1	0	0	0 (0)	0 (0)	1
MobileDeviceUpdater	11	0	0	0 (0)	0 (0)	3
mtmfs	2	0	0	3 (3)	0 (0)	1
mysqld	1	0	0	1 (1)	0 (0)	1
nehelper	1	0	0	0 (0)	0 (0)	1
nesessionmanager	1	0	0	0 (0)	0 (0)	3
NetAuthSysAgent	1	0	0	0 (0)	0 (0)	1
netbiosd	22	0	0	0 (0)	2 (2)	2
netsession_mac	1	0	0	1 (1)	2 (2)	1
networkd	1	0	0	0 (0)	1 (1)	1
networkserviceproxy	21	0	0	0 (0)	0 (0)	1
nfcd	2	0	0	0 (0)	0 (0)	1
node	2	0	6	2 (2)	0 (0)	4
Norton	2	0	0	0 (0)	0 (0)	2
Notes	10	0	0	0 (2)	0 (0)	1
NotificationCenter	20	0	0	0 (0)	0 (0)	1
nsurlsessiond	22	0	11	0 (10)	0 (0)	2
nsurlstoraged	20	0	0	0 (0)	0 (0)	1
ntpd	3	0	1	0 (0)	1 (8)	1
ocspd	1	0	0	0 (0)	0 (0)	2
Office365ServiceV2	4	0	0	1 (1)	0 (0)	2
OneDrive	1	0	0	0 (6)	0 (0)	4
opendirectoryd	1	0	0	0 (0)	0 (0)	1
OPERA	2	0	41	0 (14)	0 (5)	27
OSDUIHelper	15	0	0	0 (0)	0 (0)	1
p	1	0	2	0 (2)	0 (0)	2

P72E3GC48.com.dashlane.Dashlane	1	0	10	1 (1)	0 (0)	0
Pages	2	0	0	0 (0)	0 (0)	2
parentalcontrolsd	2	0	0	10 (10)	0 (0)	4
parsecd	10	0	0	0 (0)	0 (0)	1
pbs	7	0	0	0 (0)	0 (0)	1
pdftex	1	0	3	0 (0)	0 (0)	0
PerfPowerServices	15	0	0	0 (0)	1 (1)	0
photoanalysisd	4	0	0	0 (0)	0 (0)	1
photolibraryd	1	0	0	0 (0)	0 (0)	1
plugin-container	1	0	20	0 (0)	0 (0)	42
pma	1	0	0	0 (0)	0 (0)	1
powerd	1	0	0	0 (0)	0 (0)	1
POWERPOINT	11	0	0	0 (1)	0 (0)	3
Preview	18	0	0	0 (0)	0 (0)	1
PRINTING	8	0	0	0 (0)	0 (0)	2
ProtectedCloudKeySyncing	2	0	0	0 (0)	0 (0)	1
PulseTray	2	0	0	0 (0)	0 (0)	1
PYTHON	2	0	17	1 (1)	0 (0)	16
QuickLookUIService	2	0	0	0 (0)	0 (0)	1
R	3	0	8	1 (1)	0 (0)	2
rapportd	19	0	0	2 (2)	1 (1)	2
rcd	1	0	0	0 (0)	0 (0)	1
RDR CEF	2	0	14	0 (0)	0 (0)	5
ReceiverHelper	1	0	0	0 (0)	0 (0)	1
recentsd	1	0	0	0 (0)	0 (0)	1
remoting_me2me_host	1	0	18	0 (1)	0 (0)	1
ReportCrash	3	0	0	0 (0)	0 (0)	1
routined	2	0	0	0 (0)	0 (0)	1
rpcsvchost	11	0	0	0 (0)	0 (0)	1
rsession	3	0	9	3 (3)	0 (0)	1
rtcreportingd	14	0	0	0 (0)	0 (0)	1
SAFARI	11	0	2	0 (19)	0 (0)	10

SCIM_Extension	3	0	0	0 (0)	0 (0)	1
ScopedBookmarkAgent	1	0	0	0 (0)	0 (0)	1
screencaptureui	1	0	0	0 (0)	0 (0)	1
secd	20	0	0	0 (0)	0 (0)	1
sharedfilelistd	1	0	0	0 (0)	0 (0)	1
sharingd	22	0	0	0 (0)	5 (5)	6
SidecarRelay	8	0	0	0 (0)	0 (0)	1
Similar Photo Cleaner	1	0	0	0 (0)	0 (0)	1
SIRI	9	0	0	0 (0)	0 (0)	2
SiriNCService	1	0	0	0 (0)	0 (0)	1
SLACK	1	0	40	0 (3)	0 (0)	8
soagent	1	0	0	0 (0)	0 (0)	1
socketfilterfw	3	0	4	0 (0)	0 (0)	2
softwareupdated	1	0	0	0 (0)	0 (0)	2
Spark	1	0	0	0 (5)	0 (0)	2
SPINDUMP	4	0	0	0 (0)	0 (0)	2
SPOTIFY	12	0	33	1 (9)	3 (3)	10
SPOTLIGHT	20	0	0	0 (5)	0 (0)	3
ssh-agent	2	0	9	0 (0)	0 (0)	4
Stickies	3	0	0	0 (0)	0 (0)	1
storeaccountd	1	0	0	0 (0)	0 (0)	2
storeassetd	1	0	0	0 (2)	0 (0)	2
storedownloadd	1	0	0	0 (0)	0 (0)	2
storeuid	1	0	0	0 (0)	0 (0)	1
SubmitDiagInfo	1	0	0	0 (0)	0 (0)	3
sudo	22	0	3	0 (0)	0 (0)	1
suggestd	3	0	0	0 (0)	0 (0)	1
suhelperd	19	0	0	0 (0)	0 (0)	1
SymDaemon	2	0	8	0 (1)	1 (1)	16
symptomsd	22	0	0	0 (0)	1 (1)	1
syncdefaultsd	5	0	0	0 (1)	0 (0)	2
syslogd	22	0	0	0 (0)	1 (1)	2

syspolicyd	15	0	0	0 (0)	0 (0)	1
System Preferences	4	0	0	0 (0)	0 (0)	1
systemstats	19	0	0	0 (0)	2 (2)	1
SystemUIServer	22	0	3	0 (0)	5 (5)	1
tccd	1	0	0	0 (0)	0 (0)	1
Terminal	22	0	4	0 (0)	0 (0)	1
TeXShop	2	0	1	0 (0)	0 (0)	1
TextEdit	7	0	0	0 (0)	0 (0)	1
timed	19	0	0	0 (0)	1 (1)	2
TouchBarServer	1	0	0	0 (0)	0 (0)	1
trustd	8	0	0	0 (0)	0 (0)	2
usbmuxd	22	0	0	0 (1)	0 (0)	10
useractivityd	1	0	0	0 (0)	0 (0)	1
UserEventAgent	22	0	0	0 (0)	1 (1)	4
usernoted	16	0	0	0 (0)	0 (0)	1
vpnagentd	2	0	23	1 (1)	0 (0)	1
warmd	1	0	0	0 (0)	0 (0)	1
WDSecurityHelper	1	0	0	0 (0)	0 (0)	1
WeatherService	1	0	0	0 (0)	0 (0)	1
WiFiAgent	22	0	0	0 (0)	1 (1)	1
WiFiProxy	20	0	0	0 (0)	1 (1)	0
wifivelocityd	3	0	0	0 (0)	2 (2)	2
WindowServer	18	0	0	0 (0)	0 (0)	1
wirelessproxd	1	0	0	0 (0)	0 (0)	1
WirelessRadioManagerd	20	0	0	0 (0)	1 (1)	0
WORD	16	0	4	2 (3)	0 (0)	4
WunderlistHelper	1	0	0	0 (0)	0 (0)	3
xartstorageremoted	2	0	0	2 (2)	0 (0)	0
ZoomOpener	3	0	4	1 (1)	0 (0)	0
zotero	1	0	4	2 (4)	0 (0)	2

BIBLIOGRAPHY

[1]

[2]

[3]

[4] Https encryption on the web.

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

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

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

[8] Apple. darwin–xnu commit xnu–792, Apr 2005.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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

[23] Marshall Kirk. McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The design and implementation of the FreeBSD operating system*. Addison-Wesley/Pearson, 2 edition, 2015.

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

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

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

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

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

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

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

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

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

[33] Mark Russinovich. Handle, Dec 2018.

[34] 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.

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

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

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

[38] 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.

[39] 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.

[40] Amit Singh. *Mac OS X internals: a systems approach*. Addison-Wesley Professional, 2006.

[41] 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.

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

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

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

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

[46] 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.

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