Examining the Security of Local Inter-Process Communication

Brendan Leech

Advised by Prof. Peter C. Johnson

1 Introduction

Local inter-process communication (IPC) is a way for different processes to communicate with each other while on the same physical machine. This thesis will look at three different types of local IPC: network communication over localhost, UNIX domain sockets, and named pipes. localhost is the special IP address of the loopback interface that can be used to communicate within a machine using the full network protocol stack. UNIX domain sockets are similar, although much of the overhead can be cut out since the communication will always occur within the same host, creating faster communication. Communication over both localhost and UNIX domain sockets is bidirectional. Named pipes create a dedicated unidirectional communication channel within the machine where processes can be one of possibly many readers or writers, but not both.

In this thesis, I hope to examine how applications use these three forms of local IPC, and investigate the security of these forms of communication. As shown by Bui et al., the same security procedures that are standard for networked communication are not always used for local IPC [2]. Therefore, there are possible vulnerabilities in applications that use local IPC. I will look for these in two ways: first, by examining the communication to see if either the server, client, or both can be impersonated, and second, to use fuzzing software to see if there are bugs in these processes, and possibly even crashes. A crash in response to the fuzzing software would show a failure of the process to correctly parse input, which could possibly be exploited in the future.

2 Local IPC Vulnerabilities

As shown by Bui et al., using unsecured local IPC can create vulnerabilities [2]. They explain how commonly used applications, like 1Password, Spotify, and MySQL, have insecure local communication which would allow another process running on that computer to steal private information or hijack execution. By examining the messages sent between processes, Bui et al. were able to impersonate the client, server, or both for over a dozen commonly used applications. They specifically looked at named pipes and UNIX domain sockets, but this thesis additionally examines communication over localhost.

To run a process on the same machine as the victim, Bui et al. took advantage of fast-user switching and 'nohup'. Using fast-user switching on Windows, or the 'nohup' command on Linux and OS X, processes started by a user can continue to run after the user has logged out, even if another user logs in. This would allow a user to, for example, log into the guest account on a computer, start a malicious program using 'nohup', then logout and steal another user's credentials, without the victim having any idea that their computer was compromised.

Local IPC was not a new attack vector found by Bui et al. As shown in [7], named pipes have had at least four separate types of vulnerabilities. These vulnerabilities allow an attacker to hijack execution and start processes as the victim's user on the local machine. Additionally, [5] explores the security implications of various types of local IPC on Windows and provides security recommendations. They emphasize that even though the communication never leaves the local machine, all messages should be encrypted to help keep confidential data secure from possible attacks.

A password manager is one type of application that often uses local IPC to send confidential data between processes. Therefore, these applications direct a lot of effort into securing a user's passwords. Some password managers, like [1], go so far as to make fake sets of passwords to fool an attacker who has physically stolen a

victim's computer. However, this does not prevent an attacker from listening to the communication between the password manager and the browser. To fully prevent attacks that extract information from local IPC, there must be either two-way authentication for both the server and client to know who they are talking to, or an air gap to physically separate the data [4].

3 Fuzzing

Fuzzing consists of sending random or semi-random input to a process to see how it reacts. The goal of fuzzing is to use brute force to find bugs where the software does not correctly respond to random or malformed input [6]. Many responsible software developers fuzz their own software before shipping, to help find as many bugs as possible before the software is publicly available. However, fuzzing can be done by third-parties as well, either to improve the software or find bugs that could be exploited. This type of fuzzing is more difficult since, without the source code, it is impossible to ensure that every execution path has been covered [3].

4 Proposed Methods

The first step of this project is to identify what applications to investigate. To do so, I will send out a "survey" to as many people as I can, which will create 12 data files. There will be files that describe all of the open UNIX domain sockets, pipes, named pipes, and both TCP and UDP connections. There will also be files that show how many of each of the above are open for each application. For example, Dropbox may have 55 UNIX domain sockets and 17 pipes open at the same time. I will also collect all open pipes using the 'find' program and anonymized ps output to pair up helper programs like 'Electron' with the user application that they are supporting. This data is automatically sent to a directory called 'ThesisData' on basin once the program is finished running.

Once this process is done, I will choose the applications to investigate based on the results of the survey. I will process the data that I have received and will find the applications that are most commonly used on people's computers, and the types of local IPC that each one uses. This will inform my decision so that I am investigating applications that are used often and that have different local IPC footprints. Then, I will begin the two-pronged approach of studying the messages that processes send to each other and fuzzing the communication endpoints. To see the communication over TCP and UDP, I will use Wireshark[1]. To view named pipes, I can open the reading end of each named pipe and see all communication being sent. Finally, for UNIX domain sockets, I can use one of two methods. I can sniff the packets as they are sent, either by using strace[2] or by using a tool like sockdump[3] that dumps UNIX domain socket data into a format readable by Wireshark. The other option is to modify the kernel functions that send and receive data over UNIX domain sockets to also output the communication to another location.[4]

To fuzz the communication endpoints, I will use state-of-the-art fuzzing software to try and generate crashes. The three fuzzers that I will use are radamsa[5], Sulley[6], and afl[7]. Radamsa is a basic fuzzer that only creates and sends input to a program. This allows it to easily be included in scripts that add additional functionality, such as checking if the fuzzed program crashed. Sulley includes more features, such as tracking failures and being able to run for days without manual intervention. Afl is the gold-standard for fuzzers because it requires little manual setup, is extremely fast, and uses fuzzing techniques that are successful on a large range of formats. However, it can only work when the source code is compiled with afl-gcc, so I will use it only for open-source applications. These fuzzers will attempt to find errors in the code that processes input over local IPC. Doing so will involve connecting these fuzzers to the write end of named pipes and sending data to localhost.

[1] https://www.wireshark.org
[2] https://unix.stackexchange.com/questions/219853/how-to-passively-capture-from-unix-domain-sockets-af-unix-socket-monitoring
[3] https://github.com/mechpen/sockdump
[4] https://sourceware.org/systemtap/wiki/WSunixSockets
[5] https://gitlab.com/akihe/radamsa
[6] https://github.com/OpenRCE/sulley
[7] http://lcamtuf.coredump.cx/afl/

References

[1] 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.

[2] 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. URL: `https://www.usenix.org/conference/usenixsecurity18/presentation/bui`.

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

[4] Ivan Homoliak, Dominik Breitenbacher, Alexander Binder, and Pawel Szalachowski. An air-gapped 2-factor authentication for smart-contract wallets.

[5] Zoran Spasov and Ana Madevska-Bogdanova. Inter-process communication, analysis, guidelines and its impact on computer security. 2010.

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

[7] Blake Watts. Discovering and exploiting named pipe security flaws for fun and profit, 2002. URL: `http://www.blakewatts.com/namedpipepaper.html`.