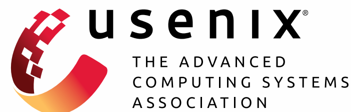


Man-in-the-Machine: Exploiting Ill-Secured Communication Inside the Computer

Thanh Bui and Siddharth Prakash Rao, *Aalto University;* Markku Antikainen, *University of Helsinki;* Viswanathan Manihatty Bojan and Tuomas Aura, *Aalto University*

https://www.usenix.org/conference/usenixsecurity18/presentation/bui

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

Man-in-the-Machine: Exploiting Ill-Secured Communication Inside the Computer

Thanh Bui*, Siddharth Rao*, Markku Antikainen†, Viswanathan Bojan*, and Tuomas Aura*

* *Aalto University* † *University of Helsinki, Helsinki Institute for Information Technology*

Abstract

Operating systems provide various inter-process communication (IPC) mechanisms. Software applications typically use IPC for communication between frontend and backend components, which run in different processes on the same computer. This paper studies the security of how the IPC mechanisms are used in PC, Mac and Linux software. We describe attacks where a nonprivileged process impersonates the IPC communication endpoints. The attacks are closely related to impersonation and man-in-the-middle attacks on computer networks but take place inside one computer. The vulnerable IPC methods are ones where a server process binds to a name or address and waits for client communication. Our results show that application developers are often unaware of the risks and secure practices in using IPC. We find attacks against several security-critical applications including password managers and hardware tokens, in which another user's process is able to steal and misuse sensitive data such as the victim's credentials. The vulnerabilities can be exploited in enterprise environments with centralized access control that gives multiple users remote or local login access to the same host. Computers with guest accounts and shared computers at home are similarly vulnerable.

1 Introduction

People use personal computers (PC) for storing and processing their most critical information, such as sensitive work documents, private messages, or access credentials to online accounts. These computers and the software running on them is designed to be personal, and the focus of security engineering has therefore been on external threats from unauthorized users and from the Internet. Nevertheless, most PCs can be accessed by more than one authorized user, making them effectively *multi-user computers*. In this paper, we analyze threats from the authorized insiders. They may be coworkers, family members, or guest users with console access.

Our focus is on the security of inter-process communication (IPC), i.e. communication channels that are internal to the computer. Computer software often comprises multiple components, such as a frontend application and a backend database, which obviously need to exchange information. Many modern desktop applications also often follow the design of web software and have a separate UI component, which connects to the business logic via a RESTful API. The UI may even be implemented in JavaScript and run in a web browser.

We assume the attacker to have login access as *non-administrator* or, at minimum, the ability to keep non-privileged processes running in the background. The attacker's goal is to exploit IPC between the processes of another user. The attacks that we discover are similar to those on the open networks, but they happen inside one computer, where application developers often do not expect adversaries. We therefore use the name *man in the machine (MitMa)* to describe these attackers.

During the analysis of case-study applications, we observed that application developers have an ambiguous attitude towards local attackers and the security of IPC channels. On one hand, these threats are not given much consideration. It is quite common to cite opinions of security experts stating that attempts to defend against local attackers are futile. On the other hand, the application implementations often make some attempt to authenticate or encrypt the communication, but rarely with the same prudence as seen in communication over physical networks.

Our main contribution is to highlight the importance of the adversary model where a nonprivileged user intercepts communication inside the computer. We demonstrate its seriousness with various examples of widely-deployed applications and compromises of critical data. We show that the vulnerabilities are common and that exploiting them is not difficult. We also discuss potential

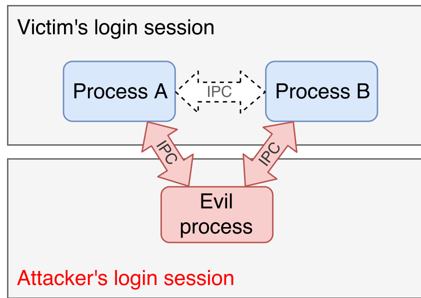


Figure 1: MitMa attack

mitigation techniques. Finally, we believe that the observations of this paper will be valuable also in the ongoing efforts to improve isolation between one user's applications.

The rest of this paper is structured as follows. Section 2 explains our adversary model. Section 3 describes IPC methods and the basic attack principles. Sections 4–7 cover the vulnerabilities found in several classes of applications. Potential solutions are covered in Section 8 while Section 9 discusses the results and Section 10 surveys related work. Finally, Section 11 concludes the paper.

2 The adversary

This section describes the adversary model and explains its relevance in everyday information systems.

We consider *multi-user computers* that may have processes of two or more users running at the same time. The attacker is a nonprivileged user who tries to steal sensitive information from or interfere with another user. It does this by intercepting communication between the victim user's processes, as illustrated in Figure 1. The malicious process is nonprivileged, and it typically runs in the background and belongs to a different login session than the victim's processes. The attack is similar to impersonation or man in the middle in computer networks, but since the communication takes place inside one computer, we call it *man in the machine (MitMa)*.

Shared computers are common both in home and enterprise environments. In a Windows domain, users are centrally registered at the Active Directory (AD) and they are typically able to log into each other's workstations. Linux and macOS workstations are commonly integrated into AD or other centralized directory services.

In addition to having its own user account, the MitMa attacker needs to be able to run a process in the background when the victim user is working on the computer. Table 1 summarizes ways to achieve this. Personal computers generally have not been designed for multiple simultaneous users, but they do support *fast user switching* [41], that is, leaving login sessions in the background and resuming them later. Such background sessions continue to have running processes that can be used in the attacks. On macOS and Linux, it is also possible to leave processes running when the user logs out (e.g., with the `nohup` command). On Windows, user processes are killed at the end of the login session, and thus the MitMa attacker must remain logged in.

MitMa attacker	**Method**	**macOS**	**Windows**	**Linux**
Authenticated user	Console login	✓	✓	✓
	SSH	✓	✓	✓
	Remote desktop	N/A	✓	N/A
Guest account	Console login	✓	✓	✓

Table 1: MitMa attackers on different OSs

The MitMa attacks can also be launched using guest accounts. The *guest user* can start the malicious process and leave the guest session in the background with fast user switching. We implemented the attacks described in this paper with macOS High Sierra, Windows 7, and Windows 8.1. These operating systems have the guest account enabled by default. Windows 10 does not currently have a built-in guest account, though creating one is possible. In enterprise Windows domains, the availability of the guest account depends on the group policy.

The attacks can also be carried out remotely, for example, if SSH [56] has been enabled. On macOS, the SSH server is started if the administrator chooses "Remote Login" from sharing preferences. Windows 10 in the developer mode also starts an SSH server. The user might not realize this because earlier Windows versions required third-party SSH servers.

Another remote access method is *remote desktop*. Non-server versions of Windows allow only one interactive session at a time. Thus, the attacker cannot access the computer at the same time as the local users. However, the remote desktop session can be left in the background and resumed later, similar to fast user switching. The MitMa attack is technically possible also between remote desktop sessions on a Windows Server. While the case-study applications considered in this paper are generally not run on Windows Server, there could be other vulnerable applications.

3 Client-server communication inside the computer

Modern operating systems (OS) provide several means for IPC. The vulnerabilities presented in this paper were found in IPC methods where a server process or device

listens for connections from client processes. Specifically, we consider network sockets, named pipes, and Universal Serial Bus (USB) communication. In this section, we give a high-level overview of these IPC mechanisms. The reader is referred e.g. to [47, 49] for more details. We also discuss the attack vectors that the MitMa attacker might exploit against each IPC type.

3.1 Network sockets

Network sockets are widely used in distributed client-server architectures. The server waits for the incoming client requests by listening on an IP address and a TCP or UDP port number. Any client can connect to the server as long as it knows the IP address and port. While network sockets were originally intended for communication across a network, they are also used for IPC within one host. If the server listens only on the loopback interface, i.e. on one of the special *localhost* addresses `127.0.0.0/8` and `::1/128`, only local client processes can connect to it.

Network sockets have almost the same functionality across operating systems. Any process, regardless of its owner, can listen on a port 1024 or higher as long as the number has not been taken by another process. Also, any local process can connect as a client to any localhost port where a server is listening. It is the responsibility of the client and server processes to authenticate each other on the application layer, as if the client was on the other side on the Internet. However, a separate connection is created for each client, and the OS prevents unauthorized processes from sniffing the communication. In that respect, IPC over the loopback interface is more secure than communication over a physical network.

Attack vectors. The malicious process, like any process on the computer, can connect to any server port on the localhost. This makes *client impersonation* very easy. Some servers might accept only one client connection, and in that case the malicious process needs to connect before the legitimate client.

The network-socket server typically listens for TCP connections on one or more predefined ports. The attacker can find the port numbers from the application documentation or source code, if available, or with commands such as `netstat`. In *port hijacking*, the MitMa attacker binds to the port ($\geq$ 1024) before the legitimate process does. The attacker can then receive any connections that clients open to the port, enabling *server impersonation*.

The MitMa attacker naturally wants to combine server and client impersonation to a full *man-in-the-middle* attack where the attacker passes messages between the legitimate client and server. This is not always easy to do on the localhost because the legitimate server and attacker cannot both bind to the same port number. Fortunately for the attacker, many applications implement *port agility* for IPC: if the primary port is taken, they choose the next port number from a predefined list. This enables the attacker to receive client connections on the primary port and connect itself to a secondary port on the legitimate server.

Even if the application uses one fixed port for IPC, the attacker may be able to *replay messages* by alternating between the client and server roles. It sometimes binds to the server port and sometimes releases it for the legitimate server. The rate of the messages passing through the attacker will be slow, but we found practical attacks that only require a small number of such role reversals.

3.2 Windows named pipes

Both Windows and Unix systems support named pipes, but the implementation details differ significantly. We describe Windows named pipes here because they were found to create more actual vulnerabilities.

On Windows, the named pipes are placed in the root directory of the named pipe filesystem. It is mounted under the special path `\\.\pipe\`, to which every user of the system has access, including the guest user. When no pipe with the given name exists, any process can create it. The named pipe can have multiple instances to support multiple simultaneous connections from clients. The creator of the first instance decides the maximum number of instances as well as specifies the *security descriptor*, which includes an access control list (DACL) that controls access to all the instances of the named pipe. The default descriptor grants read access to everyone and full access only to the creator user and the administrators. Some important details are that, if an instance of the named pipe with the same name already exists, only processes with the `FILE_CREATE_PIPE_INSTANCE` access to the pipe object can create a new instance, and that a process can set the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag to ensure that it is creating the first instance.

Attack vectors. If the named pipe is created with the default security descriptor, or with open read-write access for two-directional communication, the attacker's malicious process can connect to it and impersonate the legitimate client. The pipe server would have to configure the DACL on the named pipe object carefully to allow access for only legitimate clients.

The default security descriptor does not allow the attacker to create new pipe instances. The attacker can, however, *hijack the pipe name* by creating the first pipe instance and thus becoming the owner of the named-pipe object. This way, the attacker can impersonate the

named-pipe server. Furthermore, the attacker can set the access control list so that it allows the victim (or anyone) to create new pipe instances. If the legitimate server is careless, it will not check that it is creating the first instance of the pipe. By choreographing the creation of the instances and client connections, the attacker can then become a man in the middle between the legitimate client and server, passing messages between two pipes. It helps to know that Windows connects new clients to the server instances in round-robin order.

To summarize, it is easy to overlook the necessary security controls for named pipes, thus creating vulnerabilities, but on the other hand, careful configuration can avoid most of the issues.

3.3 Hardware security tokens

Universal Serial Bus (USB) allows peripheral devices to communicate with a computer. USB *human interface devices* (HID) include keyboards and pointing devices, but also hardware security tokens.

In Linux, HIDs are character devices and mapped to special files under `/dev/hidraw*`. The currently logged-in user gets by default read-write access to the special file. If the user session is interrupted, either by the user logging out or by switching users, the read-write access is reassigned to the display manager and later to the next logged-in user. Thus, exactly one user at a time has access to a USB HID. Windows lacks such mechanisms for dynamic access-rights assignment, and more than one user at a time could have access to a HID device including hardware security tokens.

Attack vectors. The MitMa attacker in Windows can access USB HIDs plugged in by other users. This also applies to USB security tokens. The security of the token will then depend on application-level security mechanisms implemented in the hardware or software.

3.4 Safe IPC methods

It is worth noting that some IPC mechanisms, such as anonymous pipes and socket pairs, are not vulnerable to our attacks. In these methods, both endpoints of the IPC channel are created at the same time by the same process, which prevents an untrusted process from getting to the middle. Unfortunately, these IPC methods can only be used between related processes (typically parent and child), which severely limits the software architecture. Thus, it is attractive to use the more client-server oriented but less safe methods described above.

On macOS, apart from the same IPC methods that are available on Windows and Linux, there are also Mach IPC methods that are based on the Mach kernel, such as CFMessagePort. These IPC channels are associated with a login session [10], and a process from one login session cannot interact with another. Thus, these IPC methods are immune to MitMa attacks between users.

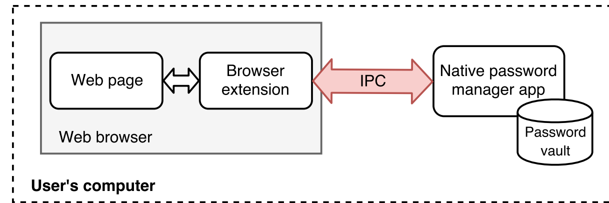


Figure 2: Password manager architecture with native app and browser extension

In the following sections, we show how the attack vectors described above are affecting real-world applications. Table 2 summarizes the applications and the vulnerabilities that we found.

4 Case study 1: Password managers

We chose password managers as our primary case study because the information they send over IPC is obviously critical and, thus, it is easy to identify security violations.

Password managers help users to choose and remember strong passwords without reusing them [24]. They store passwords along with the associated hostnames and usernames in an encrypted password vault. The key to the vault is typically derived from a master password. Password managers are often integrated to the web browser and assist the user both by offering to create and store passwords and by entering them into login pages. We focus on password managers that consist of two discrete components: a stand-alone *app* for managing the password vault and a *browser extension* for the web-browser integration, as in Figure 2. We analyze the inter-process communication between these two components.

As the following sections will show, the MitMa attacker is able to capture passwords and other confidential information from a large number of password managers. What we find interesting is that, in almost all cases, the software developers have taken some measures to authenticate or encrypt the communication between the browser extension and the app. This shows that they do not fully trust the security of the chosen IPC method. Yet, none of the studied examples implements well-designed cryptographic protection that would completely protect the communication from the MitMa attacker. The main message of the current paper is to highlight this ambivalent attitude towards IPC security and to suggest a rethink.

Application type	Application version	Browser, extension version	macOS	Windows	Linux	Communication channel	Attacks
Password managers	RoboForm 8.4.4	Chrome, 8.4.3.6 Firefox, 8.4.3.4 Safari, 8.4.5	✓	✗	N/A	Network socket	Client impersonation
	Dashlane 5.1.0	Chrome, 5.5.3 Firefox, 5.5.3 Safari, 5.5	✓	✓	N/A	Network socket	Server impersonation
	1Password 6.8.4	Safari, 4.6.12	✓	✗	N/A	Network socket	Server impersonation
	F-Secure Key 4.7.114	Chrome, 1.0.0.3 Firefox, 1.0.3	✓	✓	N/A	Network socket	Client impersonation Server impersonation
	Password Boss 3.1.3434	Chrome, 3.1.3434 Firefox, 3.1.3434	✗	✓	N/A	Named pipe	Man-in-the-middle
	Sticky Password 8.0.4	Chrome, 8.0.12.120 Firefox, 8.0.12.130 Safari, 8.0.2.63	✓	✗	N/A	Network socket	Client impersonation Server impersonation
Hardware tokens	FIDO U2F Key	—	✗	✓	✗	USB	Unauthorized access
	DigiSign 4.0.12.5850	—	✓	✓	✓	Network socket	Client impersonation
Backends with HTTP API	Blizzard 1.10.1.9799	—	✓	✓	N/A	Network socket	Client impersonation
	Transmission 2.93	—	✓	✓	✓	Network socket	Client impersonation
	Spotify 1.0.73.345	—	✓	✓	✓	Network socket	Client impersonation
Others	MySQL 5.7.21	—	✗	✓	✗	Named pipe	Man-in-the-middle
	Keybase 1.0.40	—	✗	✓	✗	Named pipe	Server impersonation

Table 2: Discovered vulnerabilities (✓ vulnerable, ✗ not vulnerable)

4.1 Managers with network sockets

Many password managers use network sockets as the IPC method because of its portability across operating systems and browsers and compatibility with web APIs. This section discusses the MitMa vulnerabilities found in such implementations.

4.1.1 RoboForm

The RoboForm [7] password manager (S) and its browser extension (E) communicate via the loopback network interface with HTTP without any authentication. The protocol is basically as follows:

1. $E \rightarrow S$: “list”
2. $E \leftarrow S$: $[item_id_1, item_id_2, ..., item_id_n]$
3. $E \rightarrow S$: “getdataitem”, $item_id_i$
4. $E \leftarrow S$: $item_i$

The extension first requests a list of all items stored in the password vault by sending an HTTP POST request to `http://127.0.0.1:54512`. The server replies with the item identifiers, which consist of a type (e.g. password, safenote) and name. To retrieve an item, the extension sends a `getdataitem` request to the server, which returns the item data in plaintext.

Attacks. Since there is no authentication between the browser extension and the native app, a MitMa attacker can impersonate the browser extension by simply connecting to the above URL. It can then retrieve all the sensitive information from the user’s password vault.

4.1.2 Dashlane

Dashlane [3] has two modes of operation: in one, the browser extension retrieves passwords directly from a cloud storage and, in the other, from a desktop app. We only consider the latter operating mode. The Dashlane app runs a WebSocket server on port 11456.

The WebSocket communication between the Dashlane app (server) and the browser extension (client) is protected as follows. First, all messages are encrypted with keys derived from a hard-coded constant secret and a nonce, which is fresh for each message and included in the message. Second, the server verifies the browser-extension id in the HTTP `Origin` header of each request. Third, the server verifies the client process by checking its code-signing signature using APIs provided by the operating system. The process must be a whitelisted web browser and the signature must be generated by a whitelisted software publisher. Fourth, the server checks that the client process is owned by the same user as the server.

A peculiar feature of Dashlane is that the browser extension collects all DOM elements from the web pages

that the user visits and sends them to the app for analysis. The app then instructs the extension on actions to take, such as to save the contents of a web form to the app when the user submits it.

Attacks. The verification of the browser binary and user id prevented us from impersonating the web browser or browser extension. However, it does not prevent impersonation of the server to the browser extension. We explored what the MitMa attacker can achieve with server impersonation. The attacker first needs the shared constant secret, which it can find in the JavaScript code of the browser extension. The attacker then runs its malicious WebSocket server on port 11456 before the benign server starts, which effectively forces the benign server to fail over to another port (the user is not notified about this). Since the attacker knows the encryption keys, the browser extension will happily communicate with the malicious server. As a result, the attacker obtains all HTML content from the web pages that the victim visits. This includes personal data displayed on web pages, such as emails and social-network messages. Furthermore, the malicious app can instruct the extension to collect web-form data and send the data to it. Then, any usernames and passwords that the user types in are sent to the malicious app regardless of whether the user wants to save them to the vault or not.

4.1.3 1Password

1Password [1] app runs a WebSocket server on port 6263. The very first time when the browser extension communicates with the WebSocket server, the server verifies the client binary and user in the same way as Dashlane does. They then run the following protocol to agree on a shared encryption key.

1. $E \rightarrow S$: "hello"
2. $E \leftarrow S$: *code* (random 6-digit string)
3. $E \rightarrow S$: *hmac_key*
4. Both the browser and the app display the *code*. The user compares the codes and confirms to the app that they match. Otherwise, the protocol restarts.
5. $E \leftarrow S$: "authRegistered"
6. $E \rightarrow S$: $nonce_E$
7. $E \leftarrow S$: $nonce_S$,
   $m_S = HMAC(hmac_key, nonce_S || nonce_E)$
8. $E \rightarrow S$: $m_E = HMAC(hmac_key, m_S)$
9. $E \leftarrow S$: "welcome"

Finally, both sides derive the encryption key $K = HMAC(hmac_key, m_S || m_E ||\text{"encryption"})$, which will be used to protect all future communication.

Attacks. The above protocol is clearly not a secure key exchange. The checks on the client binary and user, however, protect against many attacks that otherwise could exploit the protocol weaknesses. The remaining critical flaw is that the protocol requires user confirmation only on the app side. This allows the attacker's malicious background process to skip the confirmation step, and the browser extension will happily connect to it.

Because of the above flaw, the attacker can impersonate the app to the browser extension, like in Dashlane. By analyzing the JavaScript code of the 1Password browser extension, we found commands that the app can issue to the extension, such as `collectDocuments`, which tells the browser extension to collect data on the page the user is visiting including the URL and data entered into web forms.

4.1.4 F-Secure Key

The F-Secure Key [4] app runs an HTTP server on the localhost port 24166. If the port is already occupied by another process, the server fails to run.

To start using the browser extension, the user needs to cut and paste an *authorization token* from the app to the extension. The secret token is then used to encrypt parts of the messages exchanged between the app and the extension, including usernames and passwords. Additionally, every message from the extension includes a hash of the token for authentication.

When the user visits a web page, the login protocol is roughly as follows.

1. $E \rightarrow S$: *page_url*, *token_hash*
2. $E \leftarrow S$: [($description_1$, $username_1$, $password_1$), ...,
   ($description_n$, $username_n$, $password_n$)]

The browser extension requests the app for password entries that match a given URL. If matches are found, the app returns their information to the extension, including a description, username, and password. The messages are JSON objects where the values are encrypted while the keys are plaintext. Each value is encrypted as a separate message. For example:

```
{ "items": [{ "title":    "<encrypted_title>",
              "username": "<encrypted_username>",
              "password": "<encrypted_password>" }]}
```

F-Secure Key requires the user to create passwords in the app, and thus confidential data mainly flows from the app to the extension. Apart from the aforementioned messages, the extension sends a periodic `health` message to the app to indicate that it is still running and a `logout` message to lock the vault, after which the user has to enter the master password to unlock the app again. Both messages have no content except for the authorization token hash.

Attacks. As we can see, the extension does not authenticate the app before sending messages. Thus, a MitMa

attacker can impersonate the app to the extension by running an HTTP server on port 24166. Thanks to the health messages, the attacker is able to capture the authorization token hash and use it later to impersonate the extension.

With the ability to impersonate both sides, the MitMa attacker can perform replay attacks as follows. In the first stage, it impersonates the app to collect as many encrypted URLs from the extension as possible. In the second stage, the attacker closes the malicious server, releasing port 24166, and waits until the user restarts the app. The attacker then connects to the app as a client and sends the encrypted URLs. In response, the attacker obtains a list of encrypted password entries. Note that the attacker cannot decrypt the passwords. However, because the values are encrypted as individual messages and the integrity of the end-to-end connection is not checked, the attacker can modify the messages and pair the encrypted password fields with the wrong encrypted URLs. In the third stage, the attacker again impersonates the app to the extension, listening on port 24166. It can do this, for example, if the user logs out and later logs back in. The attacker then responds to requests from the browser extension by replaying the responses that it received earlier, but with the mismatched passwords. Since the passwords have been matched with the wrong URLs, they get sent to the wrong websites. As described, this is just a nuisance attack but shows that data leaks are possible. More seriously, the attacker could collude with one of the websites, identify its encrypted URL at the MitMa process by correlating the timing of the encrypted message with the user's login on the colluding site, and then leak the user's passwords to that site one by one.

4.2 Managers with native messaging

Native messaging [25] is intended to provide a more secure alternative to network sockets or named pipes for communicating between a browser extension and native code. In Windows, native messaging uses named pipes with random names for its internal implementation, and in Linux and macOS, it uses anonymous pipes. This makes the communication channel immune to MitMa attacks. The native password manager app registers an executable, called *native messaging host* (NMH), with the web browser. The configuration file of the NMH can specify which browser extensions have access to it. The web browser starts the NMH in a child process and lets the browser extension communicate with it.

Native messaging can be used to implement a password manager that is only accessed through the web browser and the browser extension. It is, however, not a complete solution for communication between the browser extension and the stand-alone password-manager app of Figure 2. This is because the NMH needs

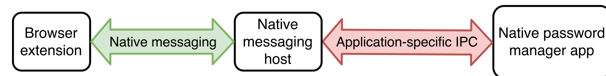


Figure 3: Communication in native messaging

to be a child process of the web browser and thus is a different process from the stand-alone app. In the following, we analyze how password managers nevertheless try to make use of native messaging.

4.2.1 Password Boss

Password Boss [6] on Windows uses both native messaging and named pipes, as shown in Figure 3. When the native app is started, it creates a named pipe with a fixed name and maximum 50 instances. The access control list on the named pipe allows all authenticated users to read and write to its instances. The native messaging host connects to the named pipe as a pipe client and forwards messages between the browser extension and the native app. Messages are sent in plaintext and no attempt is made to authenticate them.

Attacks. Any authenticated user on the system can perform the MitMa attack as follows. First, the attacker connects as a client to the native app's named pipe instance. The attacker then creates another instance of the named pipe, which is possible thanks to the unnecessarily high maximum number of instances. When the native messaging host tries to communicate with the native app, it will connect to the attacker's instance because it is the only one available. The attacker can thus sit between the two pipe instances forwarding messages and reading their content, including passwords.

The above attack does not work if the attacker only has guest access to the victim's system because the named pipe's security attributes allow only authenticated users to create and access instances. To overcome this limitation, the guest-user attacker needs to hijack the pipe name as described in Section 3. That is, the attacker has to create the first instance of the named pipe, so that it can set the DACL to allow access by everyone. After that, the guest user can mount the MitMa attack.

4.2.2 Sticky Password

Sticky Password [8] also makes use of both native messaging and WebSocket, but in a configuration that is slightly different from Figure 3. When the browser extension starts up, it uses native messaging to obtain an *AccessKey* from the NMH, which gets it from the stand-alone Sticky Password app with the CFMessagePort IPC method. After this, the browser extension communicates directly with the app's WebSocket server on port 10011.

A simplified version of the protocol between the Sticky Password browser extension and app is shown below.

1. $E \rightarrow S$: "authenticate", *ClientID*, *AccessKey*
2. $E \rightarrow S$: "GetCompleteWebAccounts"
3. $E \leftarrow S$: $[(id_1, username_1), ..., (id_n, username_n)]$
4. $E \rightarrow S$: "GetLoginPassword", id_i
5. $E \leftarrow S$: *password*

Thus, the browser extension first authenticates to the server with the *AccessKey*. It then uses further commands to retrieve the list of available data and, finally, the desired data item. Different commands exist for different types of user data.

Attacks. The first attack that a MitMa attacker can do with Sticky Password is to impersonate the WebSocket server. The reason is that the extension does not authenticate the app. That is, the attacker can hijack the localhost port 10011 before Sticky Password starts and pretend to be the app. By impersonating the server, the attacker may be able to capture data that the extension sends to the app, including new passwords that the user is attempting to save to the password vault.

Another important piece of data that the attacker can obtain with the above attack is the *AccessKey*. Once the attacker has learned this, it can impersonate the extension to the authentic Sticky Password app. That is, after capturing the *AccessKey*, the MitMa attacker closes the server socket at port 10011 and waits for the user to restart the Sticky Password app. It can then connect to the app and use the *AccessKey* to retrieve all of the victim's passwords. The attacker has to resort to this two-stage attack because, when the attacker's binary binds to port 10011, the Sticky Password app fails to do so. Nevertheless, a patient attacker is able to alternate between the connections.

5 Case study 2: Hardware tokens

Our second case study is communication with physical authentication devices. Communication with the physical tokens also takes place within one computer, and we find that it is vulnerable to MitMa attacks by malicious processes that are running in the background.

5.1 U2F security key

FIDO U2F [23] is an open authentication standard that enables strong two-factor authentication to online services with public-key cryptography and a USB hardware device called *security key*. It is supported by major online service providers, by UK government services [28], and by the Google Chrome and Firefox (beta) browsers. We analyze the security of U2F in Windows computers.

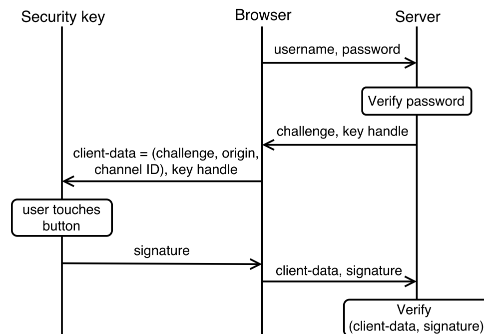


Figure 4: The basic authentication flow of a website with U2F security key [57]

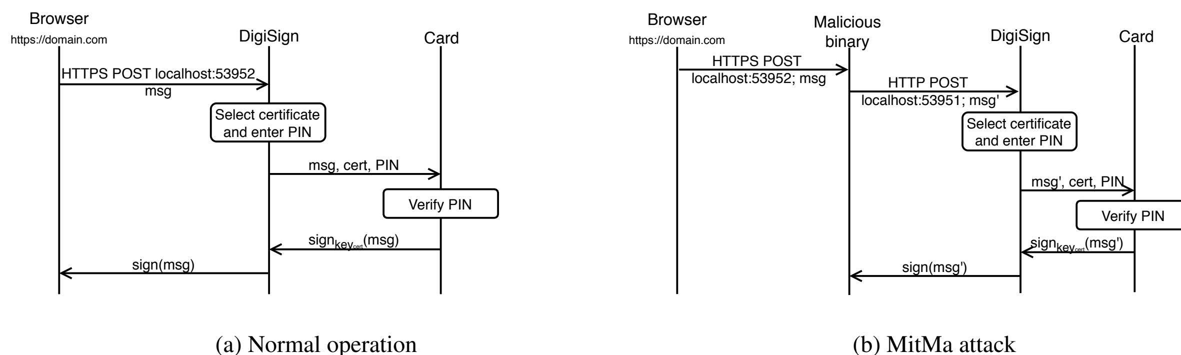
The user must first register the U2F device to the online service. The device generates a service-specific key pair and stores it together with a key handle (i.e. identifier) and the origin URL of the service.

Figure 4 illustrates the two-factor authentication process to a website. The browser receives a challenge together with a key handle from the web server. It forms the so-called `client-data` object and sends the object to the U2F device for signing. At this point, the user needs to activate the device by touching a button on the device. The browser then delivers the signed object back to the web server for verification.

The button press is meant to prevent unauthorized use of the hardware device. In practice, the browser process keeps sending signing requests to the USB device until it receives a signature back. When the button is pressed, the device responds to the first received signing request. The origin URL is included in the signed message to prevent replay attacks between websites.

Attacks. The two-factor authentication is supposed to prevent login even when the user's password has been compromised (e.g. because of attacks described in Section 4). Thus, we only consider how the attacker can subvert the U2F hardware-device authentication. To do this, the MitMa attacker creates a malicious (browser) process that runs on the user's computer and tries to log into one of the user's online services. The attacker's process then sends client-data objects to the U2F device at a high rate. When the user decides to log in to any service using U2F authentication and touches the button on the device, there is a high probability that the attacker's request will be signed. The user may notice that the first button press had no effect, but such minor glitches are normal in computers and typically ignored.

In experiments with FIDO U2F Security Key, our malicious Python client in the background was 100% suc-



(a) Normal operation

(b) MitMa attack

Figure 5: MitMa attack on DigiSign smart card reader through SCS interface

cessful in snatching the first button press and spoofing the second authentication factor to services such as Facebook and GitHub. The high success rate is due to the legitimate user's browser being slower to issue the signing requests to the device than our frequently-polling malicious script.

There are two root causes to this attack. First, the device does not have a secure path for informing the user about which request will be signed. Second, Windows allows even non-interactive processes to access the USB device in the background. This attack is not possible in Linux or macOS because they would prevent the malicious background process form accessing the USB device.

Another approach to strengthening the security of critical login sessions is the TLS Channel ID [13,21]. It does this by using a public key in addition to session cookies. However, such approaches only help protect the already established session, and they do not have any effect on the security of the initial two-factor authentication which we are able to compromise.

5.2 Fujitsu DigiSign

DigiSign is a smart-card reader application developed by Fujitsu for the Finnish government. Its main user base is healthcare professionals, but all citizens can acquire an electronic identity card for strong authentication to government services.

The DigiSign application implements the so-called Signature Creation Service (SCS) interface [34] specified by the Finnish Population Register Centre. We analyze the currently implemented protocol version 1.01.

The idea of the SCS interface is to allow a browser to send signing requests to the card-reader application without requiring any browser extensions. The basic process is illustrated in Figure 5(a). The card-reader app with the SCS interface has an HTTP server running on port 53951 and HTTPS server on port 53952 (during installation, the card reader app creates a self-signed certificate for the local HTTPS server and adds it to the trusted certificates). A webpage may send signature requests to the card reader by making a Cross-Origin Resource Sharing (CORS) requests on one of these ports on the loopback address. The data to be signed may be a document, a hash, or a token that is used for authentication. Once card reader app receives a signature request over SCS, it displays a UI dialog requesting the user to insert the smart card to the reader and to type in the PIN. If these are correct, the smart card signs the messages and the result is returned to the browser.

Attacks. The MitMa attack against the SCS protocol, illustrated in in Figure 5(b), is similar to those against password managers. The attacker's process hijacks the primary (HTTPS) port used by the SCS protocol. While the attacker cannot spoof the HTTPS server, an attempt to connect to it informs the malicious process that the user is about to sign something. The malicious process blocks this connection without closing it and sends a malicious signing request to the card reader app, which is listening on the secondary (HTTP) port. When the user enters the PIN, the card signs the attacker's data.

SCS specification version 1.1 [43] will fix some of the problems. Most notably, it mandates the use of TLS on the local IPC channel and specifies only one port for the card reader app. Because of this, it would appear that an attacker cannot hijack a port and simultaneously send a signing request to the card reader app. Nevertheless, the newer specification does not solve the root cause of the problem, which is that the client in the SCS protocol is not authenticated. A malicious process could opportunistically send signing requests to the card reader app and hope that the timing is right, or a confused user might enter the PIN by mistake. Moreover, the attacker could use out-of-band hints, such as insertion of the smart card or shoulder surfing, to time the malicious signature request approximately at the correct time window.

6 Case study 3: Software back-ends with HTTP API

A common application software architecture separates the application into a front-end component, which only handles user interaction, and a back-end server with an HTTP API, which often follows the REST design paradigm. We discuss three such applications that use network sockets and HTTP for inter-process communications. We show that a MitMa attacker can circumvent the commonly accepted security solutions that are supposed to prevent client impersonation in such applications.

6.1 Blizzard

Blizzard [2], a computer game publisher, provides the Battle.net desktop app for installing and updating games. The app comes with a background service called Blizzard *update agent*, which receives commands from the app and does the actual software installation. The update agent runs an HTTP server on localhost port 1120. The client first retrieves an authorization token from `http://localhost:1120/agent` and then connects to other endpoints.

The security of the update agent has received recent attention [26] because it was found that rogue web pages open in the user's browser could connect to it and issue malicious commands to take over the computer. The attack circumvented the same-origin policy in the browser with DNS rebinding [31]. The solution was to check that the `Host` header on the incoming HTTP requests is `localhost` and not something else.

We see a deeper problem behind the vulnerability: there is no access control to limit which processes can connect to the update agent, and the implemented solution trusts the client process to provide the correct information (`Host` header). We implemented a MitMa attacker client that spoofs the `Host` header and, thus, has no problems issuing commands to the update agent. This naturally enables the same kind of privilege escalation for the MitMa attacker as the earlier-reported vulnerability enabled for rogue websites.

6.2 Transmission

Transmission [9] is an open-source BitTorrent client. It includes a background service that handles all torrent-related activities. The service runs an HTTP server on port 9091 and accepts connections by default only from the localhost. The user can, optionally, set up a username and password for authenticating connections to the server. The client posts commands, such as adding, stopping and removing torrents, to the HTTP server.

This service has also been found vulnerable to DNS rebinding [27]. Again, the proposed solution of checking the `Host` header is insufficient to stop MitMa attacks because the attacker's background process can spoof the header. Moreover, the MitMa attacker can hijack the server port and capture the username and password from the client, before releasing the port and waiting for the legitimate server to start. The attacker will then have full access to the user's Transmission account.

6.3 Spotify

Spotify, a music streaming service, runs an HTTP server on the localhost port 4381 to accept streaming commands, such as playing a song. The server whitelists clients based on the `Origin` header in order to allow selected web pages to open in the user's browser to access the HTTP API. This access-control mechanism does not prevent MitMa attacks. The reason is that the MitMa attacker can lie about the `Origin` hostname. The attacker can then the disturb the victim by telling the server to play arbitrary songs.

7 Other client-server applications

This section will analyze two more client-server applications that make use of named pipes for the IPC.

7.1 MySQL

MySQL server on Windows can be configured so that the clients connect to it using named pipes. This may be more efficient than TCP when the client and server are on the same host [39]. The MySQL server simply creates a named-pipe instance with the name `MySQL`. This named pipe allows everyone to connect to it with full access. When a client connects, a new instance is created to wait for the next client.

Attacks. The MitMa attacker can perform a man-in-the-middle attack on MySQL connections as follows. Suppose that the server has started and it has created the first instance of the named pipe. First, the attacker creates another instance of the named pipe. This is possible due to the unrestricted DACL of the pipe. The attacker then connects to the first instance as a client. Next, the MySQL server will create a new instance to wait for a new client. However, if a legitimate client now tries to connect, it will be connected to the attacker's instance because it is the oldest unconnected instance. After this, the attacker can act as the man in the middle and forward messages between the two pipe instances.

The above attack allows the attacker to read all messages between the client and the server and to modify the

SQL queries and responses. Furthermore, the attacker can inject its own queries to the session.

7.2 Keybase

Keybase [5] is an open-source messaging app with end-to-end encryption, which is available for both phones and desktop computers. On the latter, the Keybase app has a client-server architecture. The app launches a background process that handles all of the application's tasks, such as encrypting and sending messages.

On Windows, the client accepts commands from the user and sends them to the Keybase background process over a named pipe. The background process creates the pipe with the name `keybased.sock` at startup. The named pipe's access control list grants full access for the current active user and administrators, while other users have only read access. Also, the pipe is created with the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag. Thus, the background process will not start if the named pipe already exists.

To use Keybase on a new device, the user must first sign in to the Keybase background process with his Keybase credential and then approve the new device from a previously registered device. After that, the Keybase background process on the new device has full access to the user's Keybase account.

Attacks. We see that the MitMa attacker cannot set itself between legitimate client and server because of the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag. There is also no point for the MitMa attacker to impersonate the client without having write access, which is required for two-directional communication. However, the attacker can impersonate the Keybase background process to the client by starting it before the legitimate one. This causes the legitimate background process to fail silently. Since the Keybase is open source, the attacker can simply modify the Keybase source code so that the named pipe allows full access from everyone. The attacker then runs the modified service in the background and waits for the victim's first login. When the victim signs in, approval is given to the malicious Keybase instance instead of the intended one.

8 Mitigation mechanisms

In this section, we discuss potential prevention and mitigation mechanisms for the MitMa threats. The goal is to present a taxonomy that brings order to the concepts, rather than to cover all technical details.

Spatial and temporal separation of user sessions. MitMa attacks are performed by leaving a malicious process running in the background when the victim logs in to the system. The most straightforward countermeasure is to limit the number of users that have access to each computer. Ideally, each computer would be personal to one user. If that is not feasible, the administrator of a multi-user system may implement the principle of least privilege so that users can only log into the computers that they really need to access. This includes disabling the guest account.

A slightly less drastic solution is to enforce *temporal separation*, i.e. to allow only one user's processes to be running on the computer at any one time. On Linux and macOS, this requires disabling fast user switching and remote access and killing any rogue processes that might have been left behind. On Windows, disabling user switching is not effective because the attacker can easily bypass it, for example, with the built-in Windows system tool `tsdiscon`. Instead, the *Shared PC* mode [54] should be enabled, which prohibits multiple simultaneous login sessions.

Security-conscious users can also take some protective measures by themselves. They can manually verify that there are no other active login sessions in the background, e.g. with the Windows command `query user`. The most reliable way is to reboot the computer before logging in, so that any active user sessions and processes are flushed out. Naturally, these measures help only if all remote access methods, such as SSH, have been disabled.

Access control. The developers of IPC applications should make use of OS access-control features such as Unix permissions or Windows DACLs on named pipes. Unfortunately, operating systems do not provide similar access controls for network sockets. As we have seen, access control for USB communication in Windows is also lacking. Furthermore, the cases studies in this paper show that it is easy to make mistakes with access control. For example, when creating a named pipe on Windows, the server needs to specify the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag or check after the creation who is the owner of the securable pipe object. Any checks made before the pipe creation are not reliable because of possible race conditions.

Attack detection. Once a named IPC channel has been created, the communicating endpoints can use operating-system APIs to check whether they are communicating securely with the correct entity. With Windows named pipes, the client and server can query the session and process identifiers of the other endpoint. This makes it possible to check that the client and server are in the same login session. Based on the process id, they can query further attributes of the process at the other end of the pipe, such as the user and the path to the process binary, which can then be compared to a whitelist. The critical

trick here to perform the checks both at the server for the client and at the client for the server.

JavaScript clients running in a web browser, including browser extensions, pose special challenges for such attack detection. First, they do not have access to OS APIs and are therefore unable to perform most checks on the server process to which they connect. This limitation means that it is difficult to establish secure communication between a web browser extension and a stand-alone app. Second, web browsers are highly scriptable. As we have seen, some IPC servers check that the client binary is a signed version of a well-known web browser. This check alone is not reliable because the attacker could be using the legitimate binary for malicious purposes. At minimum, the server should check the owner of the client process.

Cryptographic protection. Authentication methods for communication over insecure channels have been studied widely [15,18,30] and can be applied also to IPC. These protocols require distribution of shared or public keys to the IPC clients and servers. For example, F-Secure Key authorizes access to the password database by transferring a secret token to the client through a user-assisted out-of-band channel (in this case, Windows clipboard, which has its own weaknesses). Lessons for more secure user-assisted pairing methods could be learned, for example, from Bluetooth device pairing and other user-assisted out-of-band authentication and pairing protocols [12,16,45].

Another approach is to assume that all IPC takes place remotely over the Internet and to use the standard TLS-based protocols for protecting it. The necessary infrastructure, including certification authorities, may be an overkill when the goal is authorization of the server and client processes rather than binding them to strong identities. Even OAuth 2.0, which defines bearer tokens for client authorization and therefore seems suitable for IPC clients, depends on certificates for authenticating the server. In any case, cryptographic protection requires careful design and, as we have seen once again in this paper, ad-hoc implementations tend to have weaknesses.

Architectural changes to software. Some password managers do not have a stand-alone app but connect directly from the browser extension to a cloud service, which provides the server functions. This kind of architecture avoids inter-process communication altogether but is not feasible for all applications.

Another way to avoid the vulnerabilities of IPC methods is to redesign software to run related software components in the same process. This does not necessarily mean loss of software modularity or use of third-party components. For example, SQLite does not require IPC in the same way as MySQL does because it is linked to the application as a library. The safe IPC methods (unnamed pipes and socket pairs, see Section 3.4) can still be used between related processes without exposing the applications to MitMa attacks.

Such architectural solutions work well when they are a good match for the goals of the application developer. In many cases, however, the developer would not be willing to give up common software patterns like separating software into a frontend UI and backend business logic and database that run on the same computer, or communicating with a web API between these components.

9 Discussion and future work

The described vulnerabilities are fundamentally caused by carelessly-designed or poorly-written software. This conclusion is supported by the fact that there are also secure, well-designed applications that make use of IPC. As a further case study, we looked at cloud-storage applications (e.g. Dropbox, SpiderOak, Box), which tend to have a local backend component that is accessed over IPC. We found this class of software to be more prudent about security than the ones discussed in this paper. Because of such positive examples, our view of the future is not entirely bleak.

The well-designed applications set up strict DACLs or permissions to ensure that the IPC channel is accessible only to the authorized user(s) and configure the IPC channel options carefully rather than relying on the default settings. They also query the OS APIs to check that the login session, user and executable file of the other endpoint have the expected values. Named pipes provide more such control and seem easier to secure than network sockets. The advantage of network sockets is that the same web APIs work without code changes locally and across the Internet, but the cost is that the available web security mechanisms do not take advantage of the locality and are usually considered too heavy for local IPC.

The explanation why the problems with IPC are so widespread is probably twofold. First, developers are inclined to consider the localhost a trusted environment. Second, the best practices for secure IPC are not documented, and therefore developers may simply be unaware of the threats and solutions. We therefore believe that the best way to address both of these potential explanations is to raise awareness about the attacks and defenses, as we attempt to do in this paper. Over time, better tools such as safe APIs and security test benches could help eradicate entire classes of problems. Fully automated vulnerability scanning, however, does not seem possible because the automated tools cannot not evaluate the security of application-level cryptographic protection.

In some sense, the idea of protecting the users of a multi-user computer system from each other takes us back to the early days of computer security. With personal computers, this has not been perceived as so important. It has also become common wisdom among information-security experts that, if the attackers can run a process on the computer, they always can find a path to *privilege escalation* [32, 44, 55] and gain full administrative access. There is, however, the opposite trend towards greater isolation of applications from each other and containing malicious applications. This trend started in mobile devices, but desktop operating systems are beginning to provide similar protections (UWP AppContainers in Windows 10 [40] or application sandboxing in macOS [11]). The MitMa attacks are one way for a non-privileged process to circumvent isolation boundaries within the computer, and we believe that the observations of this paper will prove useful in the design of application-isolation mechanisms.

We have focused on the threat model where the attacker and victim are two nonprivileged users. One direction of further work is to look at similar MitMa vulnerabilities in server software where a non-administrator attacker exploits IPC for privilege escalation. Attacks between applications of the same user may also deserve a look. Even though current desktop applications will not present much resistance to such attacks, it is good to question the status quo. Such threats have earlier been studied in the context of Mac OS X [55] and mobile OSs [22,46,55], which, as mentioned above, already provide isolation for user-space apps.

10 Related work

This section summarizes the research literature related to the attacks presented in this paper.

IPC security. Windows named pipes have been an attractive target for security analysts. Even though the OS offers security controls to named pipes using DACL, the default security descriptor of a Windows named pipe allows anyone to read its content [38]. In some cases, there could be write access for everyone due to the developer's negligence. In such scenarios, even a remote attacker may be able to impersonate the pipe client to perform code execution or denial of service [19, 20]. The server-impersonation and name-hijacking attacks explained in our paper are not feasible for such remote attackers.

Additionally, named pipes are also known to be vulnerable to an impersonation attack [53] (unrelated to the client or server impersonation of our paper). The pipe server impersonates its client's security context, which allows it to perform actions on behalf of the client. This attack requires the server and client processes to run as the same user or for the server to run as the superuser, which is a stronger assumption than our threat model.

Vulnerabilities have also been found for other IPC mechanisms. Xing et al. [55] demonstrated that a malicious application on macOS and iOS can access another application's resources despite the app isolation. The attacks intercept IPC in a way similar to ours, but the malicious binary is executed with the victim's privileges. Related problems have also been found in Android app isolation [22, 46].

The DNS rebinding vulnerability [26, 27, 31] that we referred to in Sections 6.1 and 6.2 has simple solutions based on whitelisting. It is, however, known that whitelisting approaches, such as cross-origin resource sharing (CORS) for HTTP, often lead to the use of unsafe wildcard policies. Such too-relaxed whitelists on locally-running services may enable `XMLHttpRequest` from untrusted web applications (without the DNS rebinding of [26, 27]). These attacks are akin to our client impersonation, but the attack is launched from a supposedly sandboxed code running in the web browser rather from another user's session.

Automated detection and firewall-like defenses may help to prevent attacks between users and applications inside the same computer. Vijayakumar et al. [51] automate the detection of name-resolution vulnerabilities with dynamic analysis of software. A process firewall can prevent unauthorized cross-user resource access with system calls [52] and file and IPC squatting attacks [50]. The attacks presented in the current paper could be prevented by firewalling of applications, although it may become burdensome to whitelist the desirable interactions accurately.

Password manager security. Secure and usable integration of a password manager and a browser is a widely studied problem. Because the password manager is expected to autofill passwords into web forms, the credentials are exposed to network attackers running malicious scripts on the website. Silver et al. [48] showed that the autofill policies in some browsers allow a network attacker to steal credentials. Li et al. [37], on the other hand, found that password managers suffer from traditional web vulnerabilities (e.g. XSS, CSRF), poor user-interface design, and problems related to poorly understood threat model. Unlike the remote attacker in these publications, our MitMa attacker exploits the IPC communication within a single computer.

There have been several attempts to create more secure password-manager architectures, more specifically to address autofill attacks [36] and offline cracking attacks [17]. While they illustrate the wide variety of threats that must be taken into account when designing a password manager, to our best knowledge, there is hardly

any previous work that would address the security issues arising within the computer.

USB hardware token security. Hardware tokens can be used as a second authentication factor to protect against credential leaking, phishing, and man-in-the-middle attacks [35]. The security of the tokens has been studied under various threat models [14, 29, 33, 42]. Unlike the attacks in these papers, our MitMa attacks neither require the attacker to physically access the hardware token nor to find a side channel.

11 Conclusion

We analyzed the security of inter-process communication in the presence of a nonprivileged malicious process on the same computer. The malicious process may belong to another user that has login access to the computer or to a guest user. We found several vulnerabilities in security-critical applications including password managers, two-factor authentication, and applications that have been split into separate frontend and backend processes. While it is possible to use IPC in a secure way, we found that many applications either do not give much consideration to the security of local communication or they implement ad-hoc security measures that are insufficient. We expect the importance of IPC security to increase as operating system strive to isolate not only users but also applications from each other.

Following responsible disclosure, we have reported the vulnerabilities discovered in the research project to the respective vendors and believe that they have taken steps to prevent the attacks.

Acknowledgments

This work started from a collaborative research project with F-Secure. We are grateful to Alexey Kirichenko and others at F-Secure for their support and feedback. The research was partially funded by the CyberTrust program of DIMECC and Tekes (Business Finland) and by Academy of Finland (project 296693).

References

[1] 1Password. https://agilebits.com/onepassword.

[2] Blizzard. https://www.blizzard.com/.

[3] Dashlane. https://www.dashlane.com/.

[4] F-Secure Key. https://www.f-secure.com/en/web/home_global/key.

[5] Keybase. https://keybase.io/.

[6] Password Boss. https://www.passwordboss.com/.

[7] RoboForm. https://www.roboform.com/.

[8] Sticky Password. https://www.stickypassword.com/.

[9] Transmission. https://transmissionbt.com/.

[10] APPLE DOCUMENTATION ARCHIVE. Root and login sessions on OS X. https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPMultipleUsers/Concepts/SystemContexts.html, Apr. 2013.

[11] APPLE DOCUMENTATION ARCHIVE. App sandbox design guide — app sandbox in depth. https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandboxDesignGuide/AppSandboxInDepth/AppSandboxInDepth.html, Sept. 2016.

[12] AURA, T., AND SETHI, M. Nimble out-of-band authentication for EAP (EAP-NOOB). Internet-Draft draft-aura-eap-noob-03, IETF, July 2018.

[13] BALFANZ, D., AND HAMILTON, R. Transport layer security (TLS) channel IDs. Internet-Draft draft-balfanz-tls-channelid-01, 2013.

[14] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. Efficient padding oracle attacks on cryptographic hardware. vol. 7417 of *LNCS*, Springer, pp. 608–625.

[15] BELLARE, M., POINTCHEVAL, D., AND ROGAWAY, P. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology - Eurocrypt 2000* (2000), vol. 1807 of *LNCS*, Springer, pp. 139–155.

[16] BLUETOOTH SPECIAL INTEREST GROUP. Simple pairing, V10r00. Whitepaper, Aug. 2006.

[17] BOJINOV, H., BURSZTEIN, E., BOYEN, X., AND BONEH, D. Kamouflage: Loss-resistant password management. In *European Symposium on Research in Computer Security, ESORICS 2010* (2010), vol. 6345 of *LNCS*, Springer, pp. 286–302.

[18] BOYKO, V., MACKENZIE, P., AND PATEL, S. Provably secure password-authenticated key exchange using Diffie-Hellman. In *Advances in Cryptology - Eurocrypt 2000* (2000), vol. 1807 of *LNCS*, Springer, pp. 156–171.

[19] BURNS, J. Fuzzing Win32 inter-process communication mechanisms. In *Black Hat* (2006).

[20] COHEN, G. Call the plumber you have a leak in your (named) pipe. In *DEF CON 25* (2017).

[21] DIETZ, M., CZESKIS, A., BALFANZ, D., AND WALLACH, D. S. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *21st USENIX Security Symposium* (2012), pp. 317–331.

[22] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium* (2011).

[23] FIDO ALLIANCE. Universal 2nd factor (U2F) overview. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.html, Oct. 2017.

[24] FLORENCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *16th International Conference on World Wide Web, WWW2007* (2007), ACM, pp. 657–666.

[25] GOOGLE. Native messaging. https://developer.chrome.com/apps/nativeMessaging. [Nov. 2017].

[26] GOOGLE SECURITY RESEARCH. Blizzard update agent - JSON RPC DNS rebinding. https://www.exploit-db.com/exploits/43879/, Jan. 2018. EDB-ID 43879.

[27] GOOGLE SECURITY RESEARCH. Transmission - JSON RPC DNS rebinding. https://www.exploit-db.com/exploits/43665/, Jan. 2018. EDB-ID 43665, CVE-2018-5702.

[28] GOVERNMENT DIGITAL SERVICE. Introducing GOV.UK Verify. https://www.gov.uk/government/publications/introducing-govuk-verify/introducing-govuk-verify, Mar. 2018.

[29] GRAND, J. Attacks on and countermeasures for USB hardware token devices. In *Proceedings of the Fifth Nordic Workshop on Secure IT Systems* (2000).

[30] JABLON, D. P. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review 26*, 5 (1996), 5–26.

[31] JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. Protecting browsers from DNS rebinding attacks. *ACM Transactions on the Web (TWEB) 3*, 1 (2009), 2.

[32] JURANIC, L. Back to the future: Unix wildcards gone wild. http://www.defensecode.com/public/DefenseCode_Unix_WildCards_Gone_Wild.txt, June 2014.

[33] KÜNNEMANN, R., AND STEEL, G. YubiSecure? Formal security analysis results for the YubiKey and YubiHSM. In *International Workshop on Security and Trust Management, STM 2012* (2012), vol. 7783 of *LNCS*, Springer, pp. 257–272.

[34] LAITINEN, P. HTML5 and digital signatures: Signature creation service 1.0.1. Specification, Finnish Population Register Centre, 2015. https://eevertti.vrk.fi/documents/2634109/2858578/SCS-signatures_v1.0.1.pdf.

[35] LANG, J., CZESKIS, A., BALFANZ, D., SCHILDER, M., AND SRINIVAS, S. Security Keys: Practical cryptographic second factors for the modern web. In *International Conference on Financial Cryptography and Data Security, FC 2016* (2016), vol. 9603 of *LNCS*, Springer, pp. 422–440.

[36] LI, H., AND EVANS, D. Horcrux: A password manager for paranoids. Document arXiv:1706.05085v2, Oct. 2017. http://arxiv.org/abs/1706.05085.

[37] LI, Z., HE, W., AKHAWE, D., AND SONG, D. The emperor's new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium* (2014), pp. 465–479.

[38] MICROSOFT. Named pipe security and access rights. https://docs.microsoft.com/en-us/windows/desktop/ipc/named-pipe-security-and-access-rights, May 2018.

[39] MICROSOFT DEVELOPERS NETWORK. Choosing a network protocol. https://msdn.microsoft.com/en-us/library/ms187892.aspx, 2016.

[40] MICROSOFT DEVELOPERS NETWORK. AppContainer isolation. https://msdn.microsoft.com/en-us/library/windows/desktop/mt595898(v=vs.85).aspx, May 2018.

[41] MICROSOFT DEVELOPERS NETWORK. Fast user switching. https://msdn.microsoft.com/en-us/library/windows/desktop/bb776893, May 2018.

[42] OSWALD, D., RICHTER, B., AND PAAR, C. Side-channel attacks on the YubiKey 2 one-time password generator. In *International Workshop on Recent Advances in Intrusion Detection, RAID 2013* (2013), vol. 8145 of *LNCS*, Springer, pp. 204–222.

[43] PARTANEN, A., AND LAITINEN, P. HTML5 and digital signatures: Signature creation service 1.1. Specification, Finnish Population Register Centre, 2017. https://eevertti.vrk.fi/documents/2634109/2858578/SCS-signatures_v1.1.pdf.

[44] REDHAT. Kernel local privilege escalation "Dirty COW" — CVE-2016-5195. https://access.redhat.com/security/vulnerabilities/DirtyCow, Oct. 2016.

[45] SETHI, M., ANTIKAINEN, M., AND AURA, T. Commitment-based device pairing with synchronized drawing. In *IEEE International Conference on Pervasive Computing and Communications, PerCom 2014* (2014), pp. 181–189.

[46] SHAO, Y., OTT, J., JIA, Y. J., QIAN, Z., AND MAO, Z. M. The misuse of Android Unix domain sockets and security implications. In *2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016* (2016), ACM, pp. 80–91.

[47] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Operating system concepts essentials*. John Wiley & Sons, 2014.

[48] SILVER, D., JANA, S., BONEH, D., CHEN, E. Y., AND JACKSON, C. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium* (2014), pp. 449–464.

[49] STEVENS, W. R., FENNER, B., AND RUDOFF, A. M. *UNIX Network Programming*, vol. 1. Addison-Wesley Professional, 2004.

[50] VIJAYAKUMAR, H., GE, X., PAYER, M., AND JAEGER, T. JIGSAW: Protecting resource access by inferring programmer expectations. In *23rd USENIX Security Symposium* (2014), pp. 973–988.

[51] VIJAYAKUMAR, H., SCHIFFMAN, J., AND JAEGER, T. STING: Finding name resolution vulnerabilities in programs. In *21th USENIX Security Symposium* (2012), pp. 585–599.

[52] VIJAYAKUMAR, H., SCHIFFMAN, J., AND JAEGER, T. Process firewalls: Protecting processes during resource access. In *8th ACM European Conference on Computer Systems, EuroSys'18* (2013), ACM, pp. 57–70.

[53] WATTS, B. Discovering and exploiting named pipe security flaws for fun and profit. http://www.blakewatts.com/namedpipepaper.html. [Dec. 2017].

[54] WINDOWS IT PRO CENTER. Set up a shared or guest PC with Windows 10. https://docs.microsoft.com/en-us/windows/configuration/set-up-shared-or-guest-pc, July 2017.

[55] XING, L., BAI, X., LI, T., WANG, X., CHEN, K., LIAO, X., HU, S.-M., AND HAN, X. Cracking app isolation on Apple: Unauthorized cross-app resource access on macOS. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015* (2015), ACM, pp. 31–43.

[56] YLONEN, T., AND LONVICK, C. The secure shell (SSH) protocol architecture. RFC 4251, IETF, 2006.

[57] YUBICO DEVELOPER PROGRAM. U2F technical overview. https://developers.yubico.com/U2F/Protocol_details/Overview.html. [Oct. 2017].