

Received August 4, 2020, accepted September 1, 2020, date of publication September 4, 2020, date of current version September 21, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3021743

Phantom Malware: Conceal Malicious Actions From Malware Detection Techniques by Imitating User Activity

TIM NIKLAS WITTE 

Institute of Computer Science, Osnabrück University, 49090 Osnabrück, Germany
G DATA CyberDefense AG, Research and Development, 44799 Bochum, Germany

e-mail: wittet@uni-osnabrueck.de

This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG), and in part by the Open Access Publishing Fund of Osnabrück University.

ABSTRACT State of the art malware detection techniques only consider the interaction of programs with the operating system's API (system calls) for malware classification. This paper demonstrates that techniques like these are insufficient. A point that is overlooked by the currently existing techniques is presented in this paper: Malware is able to interact with windows providing the corresponding functionality in order to execute the desired action by mimicking user activity. In other words, harmful actions will be masked as simulated user actions. To start with, the article introduces User Imitating techniques for concealing malicious commands of the malware as impersonated user activity. Thereafter, the concept of Phantom Malware will be presented: This malware is constantly applying User Imitating to execute each of its malicious actions. A Phantom Ransomware (ransomware employs the User Imitating for every of its malicious actions) is implemented in C++ for testing anti-virus programs in Windows 10. Software of various manufacturers are applied for testing purposes. All of them failed without exception. This paper analyzes the reasons why these products failed and further, presents measures that have been developed against Phantom Malware based on the test results.

INDEX TERMS Malware, ransomware, user imitation, UI redressing, overlay attacks, BadUSB, obfuscation, behavior blockers.

I. INTRODUCTION

On the one hand, malware detection techniques become more and more powerful. On the other hand, malware obfuscation techniques become more developed [1]. In order to evade detection by signatures of antivirus software hundreds of packers/decryptors are applied [2]. As a consequence, behavior blockers analyze the execution flow of the program to find malicious actions [3]. Often, these detection techniques are based on machine learning algorithms such as a Support Vector Machine [4]. As a result, malware authors are looking for new possibilities to overcome these techniques. This situation is a mutual arms race between malware authors and anti-virus software producers [5].

The associate editor coordinating the review of this manuscript and approving it for publication was Ana Lucila Sandoval Orozco.

User interfaces are designed to reach the maximum user comfort. However, design principles responsible for comfort cause vulnerabilities which can be exploited by an attacker for bypassing malware detection techniques: This paper presents the User Imitating technique for bypassing behavior blockers by mimicking user input (such as keystrokes) to execute malicious actions. This means that software interacts with a user interface instead of a humane user. There are two variants of User Imitating presented: the overlay variant and the multiple desktops variant. Each of the variants conceals every change of the user interface induced by the simulated keystroke to prevent that the victim becomes suspicious in a different way. Examples for this are the existence of opened windows and that windows pop up. Phantom Malware applies User Imitating to hide all its malicious actions. Metaphorically speaking, a Phantom Malware is acting as an additional user

and masks its actions from the real current user. Both users are acting concurrently with the difference that the current user (human) does not notice the activity of the hidden user (Phantom Malware).

This article is organized as follows: Known attacks against user interfaces similar to the User Imitating technique are presented in section II (related work). This section covers the overlay-based banking trojan, tabjacking and the keystroke injection attack based on BadUSB. Section III explains the implementation of both User Imitating variants in C++ for Windows 10 in detail. The approach of the Phantom Malware is explored by elucidating the implementation of a Phantom Ransomware in section IV. In section V the operating system compatibility of User Imitating is discussed. Besides demonstrating the insufficiency of state of the art malware detection techniques, antivirus software is also tested in this section against a ransomware without User Imitating and a Phantom Ransomware in order to detect them. Here, both ransomware are doing the same malicious actions, although the Phantom Ransomware applies User Imitating to execute these actions. The results of this evaluation are analyzed in section VI. In addition, differences between the attacks against user interfaces mentioned in section II and User Imitating are enumerated. Section VII covers further improvements for Phantom Malware. Finally, effective countermeasures against Phantom Malware are presented in section VIII, followed by conclusions in section IX.

Malware is employed for criminal purpose such as spying out sensitive data and blackmailing (ransomware). However, this paper shall not be seen as a Phantom Malware construction tutorial for cyber criminals. Instead, it shall raise attention to the serious threat induced by Phantom Malware on the general public including operating system manufacturers. The measures against Phantom Malware, enumerated at the end of this paper, shall act as an inspiration for others to implement and develop further ones.

In summary, this paper makes the following contributions:

- Introducing the User Imitating technique for concealing harmful actions of malware as simulated user behavior. Two variants of this technique are introduced: overlay variant and the multiple desktops variant.
- Presenting basic concepts of Phantom Malware by discussing the implementation of a Phantom Ransomware.
- Explaining specific obfuscation techniques for Phantom Malware.
- Proposing suitable measures against Phantom Malware.

II. RELATED WORK

This section is used as an overview of the related work on attacks against user interfaces. The attacks are similar to the two User Imitating variants presented in the subsequent section, although the subsection A. *UI REDRESSING* (User Interface redress attack) will only present attacks based on overlay user interface components such as buttons and views (on Windows called: windows). There are no UI redressing attacks employing multiple desktops [6].

A. UI REDRESSING

UI redressing is a set of attacks based on a modification of the user interface (e.g. desktop and web page on a browser). The user shall be tricked into triggering an event unconsciously. This event enables the attacker to bypass security mechanisms [6], [7]. The following subsections present a specific attack of this set.

1) CLICKJACKING: TABJACKING

Clickjacking as a subset of UI redressing [8]. This subset consists of attacks like cursorjacking, filejacking and tabjacking. The UI is changed in a way that the user mistakenly clicks on another UI (user interface) element, such as a button, which the user did not intend [9], [10]. This subsection will cover a tabjacking attack for the Android operating system combined with a touch display. By touching on a UI element, an `onClick` event [11] is generated. As shown in Figure 1, a malicious application with only minimal rights is able to exploit the unconscious click by the user to an unintended UI element, so that the user buys an unwanted app from the Google Play Store. By default, the malicious application is not able to buy an app from the Google Play Store. The app bought by the user has been created by the attacker. Also, it is chargeable so that the attacker achieves a financial profit [12].

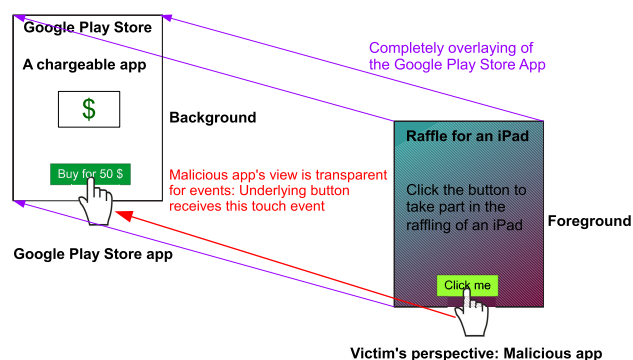


Figure 1. Tabjacking: The victim is fooled into buying an unwanted, chargeable app from the Google Play Store.

In the beginning, the malicious app is started by the user. The app creates a full screen view in the foreground. A view is a rectangular area on the screen which is responsible for event handling and drawing. Additionally, it contains UI components such as buttons and text fields [13]. The malicious app opens the preinstalled Google Play Store app in the background. The view of the malicious app overlays the opened Google Play Store app. As a result, the user does not notice that the Google Play Store app is opened. The chargeable app of the attacker is selected in the Google Play Store app. Furthermore, the Google Play Store app contains the button *Buy for 50 \$*. If the user clicks on that button (by touching on it) the chargeable app will be installed on the system. By then, the user will have already paid for the installation. The view of the malicious app contains the button

Click me. This button and the button of the Google Play Store have the same relative positions: In other words, the button of the malicious app is placed over the button of the Google Play Store. The malicious app's view is transparent for any UI events including `OnClick` [11] events. UI events will occur in the Google Play Store's view instead of the malicious app's view. Screen coordinates of this event are unchanged. As a consequence, if the user clicks on the button *Click me* the related UI event will occur on the underlying button *Buy for 50 \$*. The user unintentionally buys the chargeable app of the attacker. Afterwards, the Google Play Store app is closed. The view in the foreground will be closed by the user. The user is baited to click on the button *Click me* by fooling them to believe to be participating in a raffle for an iPad. Over and above, social engineering is applied for this attack, too [8].

Tabjacking attacks are not limited to install unwanted software. These attacks can be applied for executing actions for which there exists a UI element on an already installed app: By pressing on this UI element the action is executed. For instance, call a predefined phone number, change the account settings, give access to microphone and webcam, etc. [8], [10].

Desktop-based operating systems such as Windows are not vulnerable for the tabjacking attack. The UAC (User Account Control) on Windows prevents this attack: If the user changes the security level of the system by modifying a registry key or disabling the firewall, the UAC opens a dialog box on an additional desktop. The current desktop is set to this desktop. The user must enter their password to confirm this change. Only the UAC is able to interact with the desktop. The malicious app doesn't have the privileges to open a window on the desktop created by the UAC [14]. The icon of the window, opened by the malicious app, will appear in the taskbar. Opening a window in Windows induces a pop up. This window is presented in the foreground and has keyboard focus which makes the user perceive it immediately.

2) OVERLAY: BANKING TROJAN

This subsection is focused on Android because overlay-based banking trojans often occur on this operating system [15]. An overlay is a feature of user interfaces: A mobile app is able to place an additional view layer over another app's view layer [16], [17]. With the overlay feature, interacting with multiple opened apps at the same time shall become more comfortable for the user [18]. However, an overlay is able to intercept user input such as key events that was originally intended for the underlying view (other app). The overlay feature provides an opportunity for malicious apps (malware, here: banking trojan) to steal login data [16].

Figure 2 presents a banking trojan which applies an overlay view for stealing login credentials. The trojan waits until the user opens a specific application which requires a login. In this case, the application is a banking app. Firstly, an overlay view is created by the trojan which then is placed on top of the banking app's view. The overlay view stays in the foreground

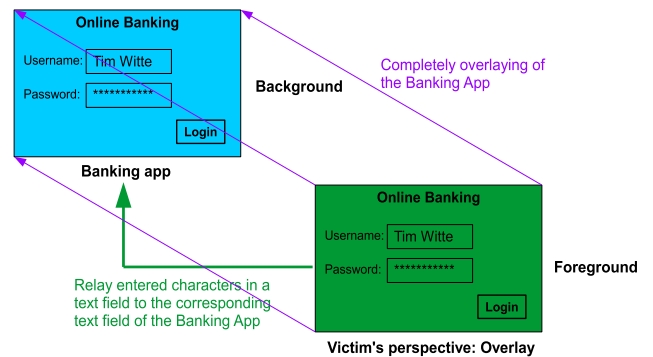


Figure 2. Overlay-based banking trojan monitoring login data.

while the banking app view stays in the background, both having the same appearance. In other words, these views have the same view properties (e.g. size, style etc.) and contain UI components such as text boxes which look alike. Over and above, both views are optically indistinguishable from each other. As a consequence, the user is fooled into believing that the overlay view on top of the banking app is the real/original one [19]. It is worth noting that the trojan is not scanning the banking app's view to copy its appearance. In fact, the view appearance was predefined during trojan creation [18].

The user enters their login data in the login fields (text fields for username and password) of the overlay view. Each character of the login credentials will be monitored by the trojan. To maintain this deception, the trojan relays each of the characters entered in a text field of the overlay view to the corresponding original text field of the banking app view. For instance, the characters entered in the "Username" text field of the overlay view are relayed to the "Username" text field of the banking app view. The relay is induced by sending key events to the banking app view's text field. If the user presses the *Login* button of the overlay view, the `OnClick` event [11] for the *Login* button of the banking app view will be mimicked. The captured login data will be sent to the attacker. Afterwards, the overlay view closes and the banking app view is placed in the foreground again. Due to the banking app view having received an `OnClick` event [11] for its *Login* button, the app will process the username and password relayed by the trojan. In hindsight, the user does not notice that their login credentials have just been stolen [19].

The overlay attack for stealing login data will not occur on desktop-based operating systems such as Windows. Instead of installing an application for the login process, the user visits the corresponding web site and logs in there. For gathering the information of the current visited website on Windows, the malware must interact with the browser by reading browser process memory. This is a malicious action. On Android, the malware (trojan) must only list all running apps and check if a specific app (here: Banking App) is open.

B. KEYSTROKE INJECTION BASED ON BADUSB

USB devices are embedded with microcontrollers which include a CPU and sometimes even a bootloader. The CPU is

executing the firmware which defines responses to requests of the host (USB controller). The bootloader enables the device to load firmware such as updates [20].

A BadUSB device is a USB device mimicking an additional, hidden (not obviously visible) USB device. This hidden device does harmful actions such as keystrokes injection caused by faked key events of a keyboard. Due to firmware modifications done by the attackers, the USB controller is fooled into perceiving the plugged USB device as a “keyboard” during its installation. However, there is no authentication process: The attackers only exploit the *trust-by-default* design principle of USB [21]. The device additionally describes itself as it “actually looks like” e.g. mass storage in case of a USB stick, preventing the victim of becoming suspicious. This, again, is a social engineering technique: The victim is convinced that their plugged-in USB stick (BadUSB device) is “only” a mass storage. The only expectation of the victim is to see a window pop up allowing them to interact with the mass storage such as transferring files on it.

Figure 3 shows the injection process of a single keystroke induced by a mimicked keyboard of a BadUSB device. As mentioned above, the USB controller is deceived into perceiving the plugged BadUSB device as a “keyboard”. USB devices are only capable of transferring data on the bus if there is an explicit request by the host (USB controller) [22]. The USB controller communicates with the “keyboard” by employing interrupt transfers which is a data transfer mode of a USB device. Further, it checks in regular time intervals (polling requests) if a key event (key state change to press or release) has occurred. The polling rate of a USB keyboard amounts to approximately 1000 Hz: Every

millisecond, the USB controller checks the keyboard for a key event. For injecting a single keystroke, a key pressed event and its corresponding released event must be mimicked. In order to emulate a key event, the “keyboard” responds to the polling request of the USB controller with this key event. The USB controller stores the data received by the “keyboard” which writes the occurred key event into its memory. Thereafter, the USB controller signals an interrupt request [23] hence, the CPU calls the corresponding Interrupt Service Routine (ISR) which was predefined by the “keyboard” driver during the “keyboard” installation process. This ISR reads out the data which decodes the key event from the USB controller [24]. Following, the operating system (here: Windows) generates internal messages [14] such as WM_KEYDOWN [25] and WM_KEYUP [26]. These keystroke messages will be stored in the System Message Queue. The system processes will then send these messages to the window which currently has keyboard focus [27]. Receiving the messages will cause a specific reaction in the window, e.g. the insertion of a character into the window’s text box.

Often, keystrokes injection based on BadUSB is used to drop malware on the victim’s machine [28]. A terminal window is opened by emulating a corresponding keyboard shortcut. Hereafter, keystrokes are simulated in order to insert the command and confirm it for execution. The command will download and execute the malware. Because of the “fake keyboard”, for the operating system it looks like the keystrokes causing these actions are induced by the user. In other words, the operating system including anti-virus software is fooled into perceiving a legit basis for these actions, i.e. that the “user” themselves is executing the actions.

Due to the polling rate of a USB keyboard, the simulated keystroke sequence is about 1000 keystrokes per second. A common way to install malware by BadUSB on the victim’s machine is to download it from a web server. Thereafter, the downloaded malware will be launched. The following command on Windows will download an (executable `.exe`) file from a web server and launch it: `powershell -command "& {iwr [download link] -OutFile [file path]} & "[file path]"` [29]. Without the download link and file path the total command has a length of 40 characters. Assuming that the download link and file path together have less than 960 characters, the total attack is finished in less than one second. Although this (one second) seems fast, the victim is able to see the popped-up terminal window and becomes suspicious. The keystroke injection attack can be disturbed if both the victim and the BadUSB device type concurrently as then, the keystrokes are mixed. The typed command in the terminal is inconsistent due to some characters typed by the user and some characters typed by the BadUSB device. An error message will be displayed and the intended command will not be executed. It may also happen that, while the BadUSB device simulates keystrokes, the user changes the window focus by clicking on another window. As a result, the simulated keystrokes take effect

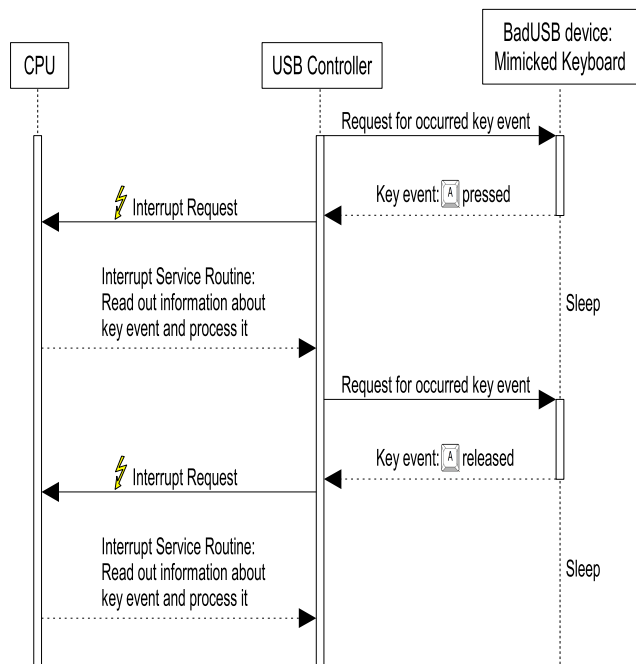


Figure 3. A mimicked keyboard by a BadUSB device injecting a single keystroke.

in the other window. For example, it is displayed in the browser's search box instead of in the terminal. In both cases, the victim would notice the attack and become suspicious.

III. USER IMITATING

Instead of applying mimicked hardware keystrokes for dropping malware, User Imitating conceals the execution of a malicious command by sending messages to an input box. As these messages represent key events for this input box, each character of the command will be inserted there. The input box is able to execute the inserted command. The corresponding window, containing the input box, must be opened beforehand, for instance Run Dialog Box (RDB), Windows Explorer (WE) etc. In the following subsections, the implementation of this technique in C++ for Windows 10 will be explained in detail.

C++ is able to interact with the WinAPI (Windows Application Programming Interface). It also provides performance improving features such as loop unrolling and function inlining [30]. Of course, also other programming language can be applied. The only requirement is that the programming language must be able to interact with the WinAPI, for instance C#. Windows is the most used operating system for desktop computers worldwide [31], Windows 10 being its latest version.

However, there are two different variants of User Imitating: Each of these variants conceals the existence of the window containing the input box and also eliminates all caused disadvantages by key event simulation mentioned in the last section in a different way.

A. OVERLAY VARIANT

The variant of User Imitating presented in Figure 4 opens a full screen window (called overlay) with the topmost (always on top) window property. Thereafter, the RDB window is opened. It contains an input box capable of command execution. After the inserted command in the input box is confirmed for execution, a CMD window opens. However, both windows will not pop up upon opening but will always stay in the background and behind the overlay. This is due to the overlay's topmost window property. In addition, the overlay contains a screenshot that had been taken before it was opened. The victim is fooled into perceiving that this screenshot is the desktop screen because they only see the full screen size overlay (desktop screenshot) in the desktop foreground. The existence of both windows is hidden from the victim.

The detailed implementation of the User Imitating variant, that is based on an overlay, is displayed in Figure 5. Firstly, a screenshot of the desktop is taken (1). The currently focused window will be identified by applying the WinAPI call `GetForegroundWindow` [32]. This window is later called actually focused window (2). Subsequently, an additional thread starts (3) which the overlay will be created in (A). It has the following properties:

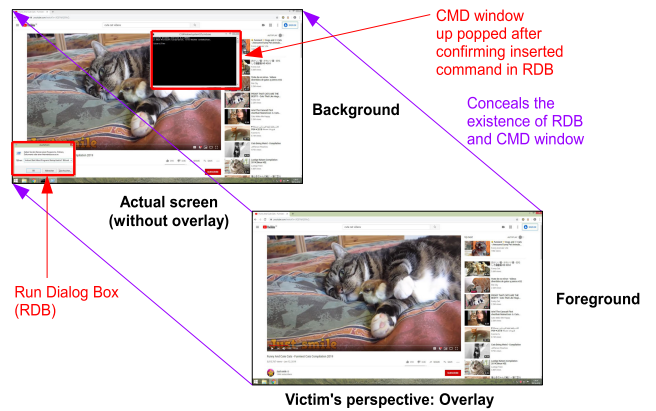


Figure 4. Overlay suppresses the displaying of the RDB window and the popped up CMD window.

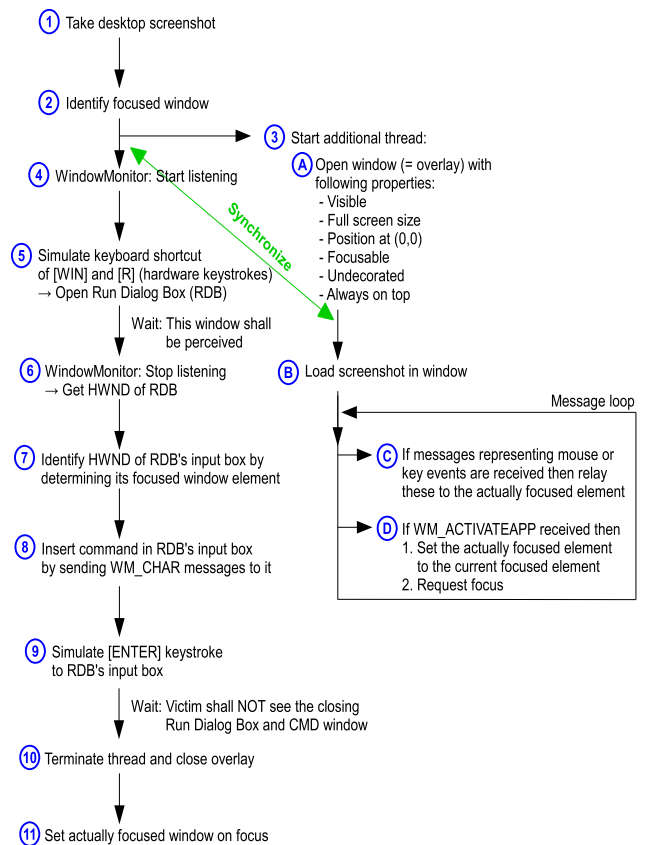


Figure 5. Implementation of the User Imitating overlay variant.

- Visible.
- Full screen size.
- Position at (0,0).
- Focusable (able to receive user input).
- Undecorated (no window title bar).
- Always on top (HWND_TOPMOST Z order state) [33].

Due to the opening of the overlay, this window is automatically brought into focus. After the screenshot was taken in step (1), it will be loaded into the overlay (B). The thread stays in the overlay's message loop: If the overlay

receives messages representing key and mouse events such as `WM_KEYDOWN` [25] and `WM_LBUTTONDOWN` [34], the messages will be relayed to the actually focused element by applying `SendMessage` [35] **(C)**. If the overlay receives a `WM_ACTIVATEAPP` [36], the actually focused element is set to the current focused element (of the focused window). The operating system Windows sends this message to those windows that have lost their keyboard focus. In order to determine the focused window, `GetForegroundWindow` [32] is applied. For identifying the current focused element of this window, `GetGuiThreadInfo` [37] is applied: The `PGUITHREADINFO` [38] is passed to `GetGuiThreadInfo` [37]. Following the function call, the `hwndFocus` value of the struct represents the current focused window element. After the actually focused element was updated, the keyboard focus is set to the overlay by using the WinAPI call `SetForegroundWindow` **(D)** [39]. Any focus changes, that occur while the overlay is open, will be detected and key and mouse events will be relayed to the new focused window element, for instance an opened window.



After the additional thread has been started, the Window-Monitor is employed: This helper object returns a list of window handles which are opened during the measure interval of the WindowMonitor. However, there must be a synchronization with the additionally created thread: The overlay window must be created in the main thread even before the WindowMonitor starts listening (beginning of its measure interval) **(4)**. Afterwards, there is a mimicked keyboard shortcut of `Win` + `R` which opens the RDB **(5)**: After the first key press there is a waiting time of about 10 ms. Then, the second key is pressed. After another waiting time of about 10 ms, both keys are released. The operating system needs time to process these key events. Nevertheless, the mentioned waiting times depend on the victim's computer speed. To imitate hardware keystrokes by software, the `SendInput` WinAPI function [40] is employed. The opening of the RDB window will bring it into focus automatically. However, the update mechanism for the actually focused element in the additional thread **(D)** ignores this window. In other words, the actually focused element must not be the RDB's input box as then, inserted command characters and characters typed by the user will be mixed up. This will cause an inconsistent command that is unable to execute. After a waiting time of about 20 ms, the WindowMonitor stops listening (end of its measure interval) **(6)**. This waiting time secures that all system processes responsible for message handling have perceived the opened RDB. The list, returned from the WindowMonitor, shall contain only one element - the window handle (HWND) of the RDB. Otherwise, the HWND of RDB will be searched for in this list by employing `GetWindowTextA` [41] for each element. The returned string is compared to *Run*, although it is language specific.

The HWND of RDB's input box will be identified by determining its focused window element **(7)**.

`GetGuiThreadInfo` [37] is applied for this determination (see above). Following, the command string will be inserted into the input box by sending a corresponding `WM_CHAR` message [42] **(8)** for each of its characters. Instead of inserting each command character in a sequence, a `WM_SETTEXT` message [43] containing the total command can be sent to this input box. Intending to execute a CMD command, the inserted string has the following form: `cmd.exe /c [CMD command]`. Another alternative is the execution of PowerShell commands. To avoid race conditions, `SendMessage` must be used for this command insertion and other outgoing messages instead of `PostMessage`. In contrast to `PostMessage`, the WinAPI call `SendMessage` does not return until the receiving window has processed the message [35], [44]. As a consequence, there is no waiting time required between the sent command characters. After the command insertion, the command is confirmed for execution by sending a `WM_KEYDOWN` message [25] with `Enter` to the RDB's input box **(9)**. The RDB window closes and the CMD window opens. The latter automatically closes after approximately 500 ms. In order to conceal the existence of this CMD window, there is a waiting time of about 500 ms. After this waiting time, the CMD window is closed. The waiting time duration always depends on the victim's computer speed. As the next step, the additional thread terminates and the overlay is closed **(10)**. The closing of the overlay causes that no window is currently focused. Finally, the actually focused window is set on keyboard focus by applying the WinAPI call `SetForegroundWindow` **(11)** [39].




Overall, the windows of the RDB and CMD are suppressed to pop up as they are forced to stay in the background by the overlay's `HWND_TOPMOST` Z order state [33]. Upon its opening, the overlay remains in the desktop foreground. It contains a desktop screenshot that had been taken before the overlay was opened. Therefore, the victim is fooled into perceiving that the overlay is the actual desktop screen and that the focused window in the screenshot, displayed by the overlay, is the actually focused window. As presented in Figure 4, the victim is fooled into thinking that the webbrowser is the focused window. The existence of the RDB and CMD windows is hidden from the victim, however, the total screen is frozen while the overlay is open. In total, the overlay is open for less than one second. In case windows are opened that contain frequently changing graphical elements such as video boxes, the victim becomes suspicious. Although the victim sees a frozen image of the video, they are able to hear the corresponding volume. The window containing the video box is suppressed to stay in the background, while the overlay window (containing a screenshot) is in the foreground. A common user behavior to a frozen screen is pressing keys to force a reaction. The overlay window catches all of these inputs and relays them to the actually focused window in the background. After the overlay window is closed, the victim notices that the keystrokes were, in fact, processed by the focused window (previously in the background). It could, for

example, be that the keystrokes are displayed in a textbox. Overall, the victim is fooled into thinking that the frozen screen is caused by a temporary fault of a system process that is responsible for window displaying. Besides, the total input capture by the overlay prevents a disturbance by the user while the command is inserted in the RDB's input box. As mentioned in the previous section, the user would otherwise be able to type in the RDB's textbox while the command is being inserted.

Instead of opening the RDB, the Windows Explorer could also be opened with the keyboard shortcut  + , although, the attack must be modified as explained below.

B. MULTIPLE DESKTOPS VARIANT

This variant of User Imitating creates an additional desktop. The Windows Explorer (WE) will be opened on this desktop. Afterwards, the command will be inserted into its input box and confirmed for execution by sending corresponding Windows Messages. Alternatively, the RDB could be employed. The total implementation is shown in Figure 6.

First of all, the additional desktop is created by applying the WinAPI function `CreateDesktopA` [45] **1**. Instead of creating an additional desktop, an already existing desktop may be used. The already existing desktop, however, must not be active (not being on focus) at that time. Accordingly, the main thread is assigned to this desktop by employing `SetThreadDesktop` [46] **2**. This thread assignment enables the `WindowMonitor` to receive information about opening windows on the additional desktop. Next, the `WindowMonitor` starts to listen **3**. To open the WE window on the additional desktop, `CreateProcessA` [47] is called: The value `lpDesktop` of the `STARTUPINFOA` struct [48] as transferred parameter is a pointer to the additional desktop's name **4**. The name was defined in step **1**. After a waiting time of approximately 20 ms, the `WindowMonitor` stops to listen **5**. The waiting time secures that system processes, responsible for window message transmission, perceive the opening of the WE window [14]. The list returned by the `WindowMonitor` shall contain only one element - the `HWND` (window handle) of the WE window. Afterwards, `SendMessage` [35] is applied to send a `WM_KEYDOWN` message [25] with  to the WE window in order to bring its input box into focus **6**. Due to assigning the main thread to the additional desktop in step **2**, messages can be sent by this thread to windows on the desktop. Following a waiting time of about 25 ms, the `HWND` of WE's input box will be determined by identifying the focused window element of the WE window **7**. This waiting time secures that system processes for desktop management [14] perceive the focus change. After the WE receives a `WM_KEYDOWN` message [25] of , it starts a "scrolling animation" by showing listed directories before its input box gets editable. According to a waiting time of approximately 1000 ms, twelve (length of default text, see below) `WM_KEYDOWN` message [25] with  are sent to this

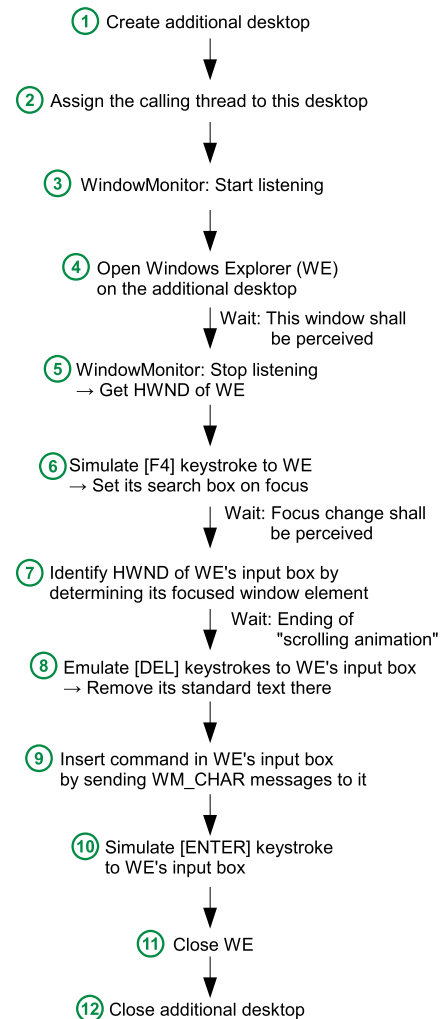





Figure 6. Implementation of the User Imitating multiple desktops variant.

input box to remove its default text **8**. In the English version of Windows 10 this text is *Quick access*. As described in the last subsection, the command string will be inserted into the WE's input box **9**. Afterwards, the inserted command will be confirmed for execution by sending a `WM_KEYDOWN` message [25] with  to the input box **10**. After the attack, the WE window on the additional desktop is closed **11**. The additional desktop will be closed by applying the WinAPI call `CloseDesktop` [49] **12**. However, steps **11** and **12** must be skipped if there is an intention to execute multiple commands in a sequence. Both steps prevent the displaying of an error message during the shutdown of the victim's computer.

The WinAPI call `SendMessage` [35] must be applied for command insertion in WE's input box and for its confirmation for execution. As the additional desktop does not possess keyboard focus, simulating hardware keystrokes by using the WinAPI call `SendInput` [40] does not work for command insertion and its execution. Also, the keyboard shortcut  and  for opening the WE window does not work.

IV. PHANTOM MALWARE

Similar to the User Imitating implementation, the Phantom Malware implementation is an exemplar presented in C++ for Windows 10. As shown in Figure 7, Phantom Malware executes all of its malicious actions by applying the User Imitating technique: A WE window is opened on the additional desktop (1). The command in the WE’s input box is inserted and confirmed for execution (2). As a consequence, the WE process launches a CMD process (3) which executes the command (4). In addition, the result of the executed command is redirected to a file by employing the > operator (5) [50]. The string, inserted in the WE’s input box, has the following form: `cmd.exe /c [command] > C:/.../output.txt`. The path to this `txt` file must be absolute. Moreover, this file must be hidden from the victim by setting its file attributes to `hidden` [51] with `attrib`.

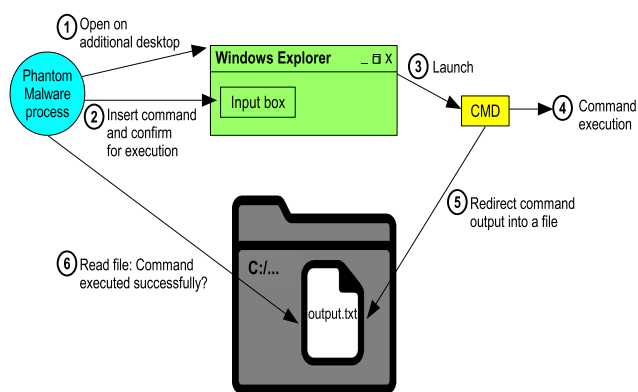


Figure 7. Overview of Phantom Malware.

The Phantom Malware reads this file by employing the WinAPI function `ReadFile` [52] and analyses its content which represents the command output (6). For example, the CMD command `move` would not return anything if the command is executed successfully. Otherwise, an error message [53] is contained in the command output. If the file (where the output of this command is redirected to) is empty, the command was executed successfully. Due to the > operator overriding the whole file content [50], the same file can be used again if there is an intention for an execution of multiple commands sequentially. Otherwise, the file must be deleted in order to cover tracks. Overall, the file is acting as a hidden information channel for the Phantom Malware, containing the outputs (results) of its executed commands. Instead of redirecting the command output to a file and reading the file to receive the information, the WinAPI call `ReadConsoleOutput` [54] can be applied. This WinAPI call is able to read the printed text (here: command result) in the popped-up CMD window, after the inserted command was confirmed for execution.

To increase the execution speed, the `&` operator [55] is applied to run multiple commands in one line at the WE’s input

box. The RDB’s input box has a capacity of 259 characters and the WE’s input box has a capacity of 2047 characters. If the inserted command in the RDB’s input box is confirmed for execution, its window closes. The WE window does, however, not close after confirming the command in its input box. Due to the increase in execution speed of the Phantom Malware, the command is inserted into the WE’s input box instead of the RDB’s input box. It is not practical for Phantom Malware to employ the overlay-based variant of User Imitating for masking all its malicious actions. If there are thousands of commands to execute (such as in the case of all victim’s files being encrypted), the victim’s screen is blocked for more than approximately two minutes: The victim is unable to interact with the windows on their desktop and becomes suspicious. However, operating systems such as Linux do not provide API calls for additional desktop creation and starting processes on this additional desktop by default - only terminal commands are provided. Hence, Phantom Malware applies the overlay-based variant of User Imitating to launch itself on the additional desktop. This point is explained in section V. Evaluation in detail.

A. EXAMPLE: PHANTOM RANSOMWARE

A Phantom Ransomware is a ransomware which employs the mentioned concept of Phantom Malware: All malicious commands are executed by applying the User Imitating technique. The output of the corresponding command is redirected to a file. It will then be analyzed if the command is executed successfully. The following subsections present the implementation of typical actions of Phantom Ransomware.

1) SET UP FOR ENCRYPTION: PREVENT FILE RESTORING

By disabling the following two system tools the victim is unable to restore its original files after encryption [56]. To disable the startup repair functionality of Windows, the command `bcdedit.exe /set {default} recoveryenabled no > C:/.../output.txt` is executed [57]. Afterwards, all shadow copies are deleted by running the following command: `vssadmin.exe delete shadows/all/quiet > C:/.../output.txt`. Shadow files are employed for restoring accidentally overwritten files [58], [59]. After each executed command, the file content of `C:/.../output.txt` must be checked for errors that may have occurred.

2) FILE ENCRYPTION

The process of file encryption by Phantom Ransomware is displayed in Figure 8. For the following section, it is assumed that the file `foo.jpg` shall be encrypted by the Phantom Ransomware.

First of all, the ransomware copies the file to another folder (here: `C:/bar`) while renaming it to `abc` by applying the command `copy C:/Pics/foo.jpg C:/foo/abc`

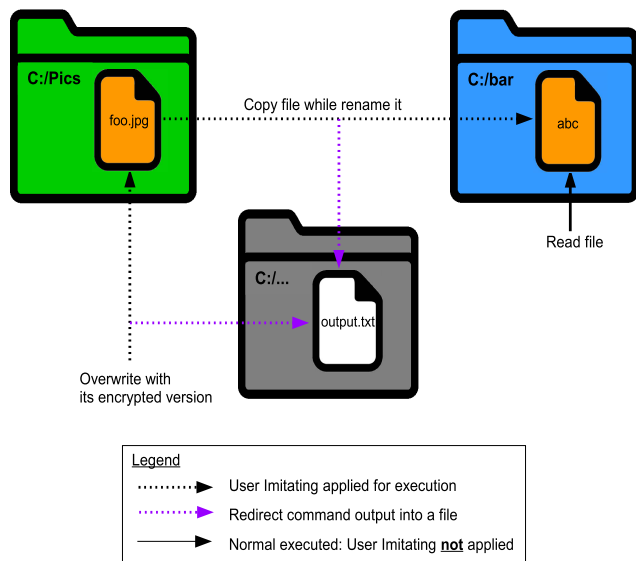


Figure 8. File encryption by Phantom Ransomware.

`/y > C:/../output.txt` Afterwards, the redirected output of the command in `C:/../output.txt` will be analyzed if an error occurred such as not having access rights.

The Phantom Ransomware reads the file `abc` without employing User Imitating. The corresponding encrypted file content will be determined. Thereafter, User Imitating is applied to overwrite `foo.jpg` with the encrypted content. To overwrite this file, the command (`echo [encrypted file content] > C:/Pic/foo.jpg`) `> C:/../output.txt` is used, although its length is not allowed to exceed the mentioned character capacity of the WE's input box. If that is the case, multiple `echo` commands must be executed. The first `echo` command is to overwrite the whole file with a part of its encrypted content by using the `>` operator. Further `echo` commands apply the `>>` operator instead of `>` [50]. In others words, each of these commands appends a part of the encrypted file content. Each output of the `echo` commands is redirected to `C:/../output.txt` and read by the Phantom Ransomware. This is done to detect any errors that may have occurred during file overwriting, for instance no file writing privileges.

Every file to encrypt is renamed to `abc` while being copied. Because the file `abc` already exists, its previous file content will be overwritten completely with the content of the copied file (in Figure 8: `foo.jpg`). User Imitating is applied for concealing this copy command and the overwriting of the copied file as user activity. As a consequence, for operating systems and anti-virus software it looks like the same file (`abc`) is read in a sequence by the Phantom Ransomware and the victim overwrites (encrypts) their own file. Before the encryption process, the file `abc` must be created with the WinAPI call `CreateFileA` [60] (no User Imitating employed) by the Phantom Ransomware. This file is directly related to the ransomware process. It is less suspicious if the

ransomware constantly reads its "own" created file rather than a file created "by the user" (User Imitating) or of another process. The file `abc` does not need a file extension as this is not required.

3) CONNECTION TO ITS C&C-SERVER

A server connection such as receiving orders or uploading data to the C&C-Server based on CMD commands is not possible [61]. Therefore, a script (here: Python) will be dropped by using the following command `echo [create socket, read data from it, print data, close socket] > C:/../GetOrders.py`. The script connects to a server, prints out the received data (or otherwise an error message) and closes the connection. At all times, the script shall be hidden from the victim. In addition, Python must be installed on the victim's machine—the existence of a Python version installed on the victim's machine must be checked and also obfuscated. Furthermore, if the script size exceeds the character capacity of WE's input box, multiple `echo` commands must be employed for its creation.

Afterwards, the python script will be launched by running the command `python C:/../GetOrders.py > C:/../output.txt`. The Phantom Ransomware receives orders from its C&C-Server by reading `C:/../output.txt`. In case of an error, the file contains an error message instead. Subsequently, the script file is deleted in order to cover tracks. Network analysis tools such as TCPView [62] are not able to trace the connection of the Phantom Ransomware with the C&C-Server. The WE is manipulated to execute the script as one of its child processes. This particular child process terminates after a very short time (about 1 ms) because its executed script only requests data from a server. The data has an approximate length of 10 bytes (decoded orders from C&C-Server).

To send data to the C&C-Server, a script similar to the script mentioned above is applied: Instead of reading data from a socket, this script writes data to a socket and prints a failure message in case an error occurs. Accordingly, the redirected output of the launched script (`send result`) in `C:/../output.txt` is analyzed on whether the data was successfully sent or not.

Instead of dropping and executing a script, it is possible to execute an inline script. Inline script means that the whole script fits into a single line as only one command. As mentioned above, its length must not exceed character capacity of WE's input box.

B. OBFUSCATION TECHNIQUES

1) ENCRYPT COMMANDS

Memory scanners of anti-virus programs dump and analyze the process memory for specific patterns (signatures) during the process run time. As a consequence, commands (including dropped scripts) in form of strings in Phantom Malware's memory will be found by these scanners. To prevent a

detection of signatures by memory scanners based on these applied commands, the commands must be encrypted. As mentioned above, the command string will be inserted into the WE's input box by sending a corresponding WM_CHAR message [42] to this input box for each of its characters. Therefore, it is possible that the total command string exists in encrypted form in memory during run time: Each character of the command string will be decrypted and sent to the WE's input box. Before decrypting the next character, the current one in memory must be overwritten. Hence, during run time there exists only one decrypted character of the command string in process memory at a time.

As an example, the command (inserted in the WE's input box) `cmd.exe /c move X Y > Z` will be encrypted during creation of the Phantom Malware and stored in program memory as a character array data structure. Nevertheless, the command arguments are file paths which vary for every computer. As a result, X, Y and Z are placeholder characters representing these file paths. The file paths are determined before the command is inserted into the WE's input box. Instead of the placeholder character, its representing file path will be inserted into the WE's input box. For each character of the file path a corresponding WM_CHAR message [42] will be sent to the WE's input box. However, these file paths do not have to be encrypted. Memory scanners usually do not apply signatures based on file paths because it is considered bad practice.

2) RANDOMIZE INSERTED COMMAND

As mentioned above, commands are connected together in one line at the WE's input box by employing the & operator [55]. To increase the effort in creating predefined patterns of behavior blockers in order to detect Phantom Malware based on inserted characters, the number of commands, connected in one line, is randomized. In addition, the connection order is randomized, although data dependencies must be noticed.

Phantom Malware can be detected by monitoring WM_CHAR messages [42] sent by a process. Based on the monitored messages, the inserted command (inserted into a text box capable for command execution such as WE's input box) will be determined and analyzed for malicious behavior. A scan of the inserted command to a harmful action is necessary in order to prevent a false positive classification as Phantom Malware for tools used by physically incapacitated people. These tools insert harmless commands into the WE's input box, for instance voice control. In order to prevent a detection by this technique, the DOSfuscation technique is applied to obfuscate each command as explained in the following. DOSfuscation hides the actual CMD command by changing its length and character sequence while the result of its execution remains unchanged [63].

Each command exists in encrypted form in the program memory. The decryption process is done during the creation of the Phantom Malware. The number and order of the commands which are connected together in a line by applying

the & operator is determined randomly. Each command is decrypted and obfuscated by applying DOSfuscation. This obfuscation is random, for instance the command will be obfuscated by inserting a random amount of ^ characters. A ^ character is an escape character in MS-DOS [61]. However, the obfuscation must not change the placement characters which represent arguments of the called command which vary from user to user (such as file paths). Furthermore, the command will be stored in memory in encrypted form. Before decrypting and obfuscating the next command, the current decrypted command must be overwritten. In other words, during command obfuscation there exists only one decrypted command in program memory at a time. Before decrypting the next command, the current command is encrypted immediately after its obfuscation. This reduces the risk to be detected by memory scanners. Next, the total command is inserted as described in the previous subsection. The inserted command is being monitored by the detection technique. Before the command is checked for malicious actions, it must be deobfuscated (converted back to the original command), although this is a time-consuming process.

3) RANDOMIZE INSERTION SEQUENCE OF COMMAND CHARACTERS

Instead of inserting each character of the command in its actual order (left to right), the Phantom Malware randomizes the order in combination with a corresponding movement of the WE's input box caret position (insertion point).

To look at an example: The text *test* shall be inserted into the WE's input box as shown in Figure 9. The order of the character insertion will be randomized as explained hereunder. Based on this order, corresponding movements of the caret position are determined. In the beginning, the WE's input box contains blank text and the caret is on the first insertion position ①. The character *e* is inserted into the input box ②. The caret of this input box will be moved back to the first insertion position (behind the character *e*) ③. A corresponding EM_SETSEL message [64] is sent to this input box, in order to set its caret to a specific position. After the letter *t* is added ④, the caret of the input box is set to the third insertion position (behind the character *e*) ⑤. Accordingly, the character *t* is inserted into the input box ⑥. WE's input box caret is moved to the third insertion position ⑦. Finally, the character *s* is added ⑧ to the input box.

As a result, the effort for determining the command, inserted into the WE's input box, is increased for the Phantom Malware detection technique mentioned in the previous subsection. Besides monitoring sent WM_CHAR messages [42] to determine the inserted command, EM_SETSEL messages [64] must be recorded.

4) SUSPICIOUS AMOUNT OF SENT WM_CHAR MESSAGES

Phantom Malware sends thousands of WM_CHAR messages [42] to mask the execution of all its malicious actions. For instance, the process of encrypting a single file by the



Figure 9. Inserting the text *test* into WE’s input box based on a random character insertion order with a corresponding movement of the caret position. The caret position is highlighted red.

Phantom Ransomware as presented in Figure 8 requires sending about 50 WM_CHAR messages [42]. In order to overwrite this file, a WM_CHAR message [42] must be sent by the Phantom Ransomware for each byte of the new file content additionally.

Nevertheless, for a process, the total amount of sent WM_CHAR messages [42] is suspicious. This is because there would not be a practical use for such an amount of sent messages. Phantom Malware could be detected by counting WM_CHAR messages [42] sent by a particular process. This process and its launched child processes share the same WM_CHAR message [42] counter with the goal to prevent a bypass of this technique. A bypass would be to apply child processes which insert only part of the command. The amount of WM_CHAR messages [42] sent by these processes is checked for if a predefined threshold is exceeded. The threshold defines a suspicious amount of sent WM_CHAR messages [42]. For expository purposes, it will be assumed in the following that the threshold is set to 40 sent WM_CHAR messages [42]. In order to evade this detection technique, the Phantom Malware sends these messages by using multiple processes as presented in Figure 10. The parent/child process relation between the Phantom Malware and its child processes is concealed by applying User Imitating.

Each of the launched processes sends messages to the WE’s input box in order to insert only a part of the total command. The final command is very long (approximately 2000 characters) because it contains commands connected together in a line at the WE’s input box by using the & operator [55] until reaching the character capacity of this input box. As mentioned in the previous subsection, the number and the order of these commands is randomized. Each of these commands will be randomly obfuscated by employing DOSfuscation. Thereafter, the Phantom Malware creates a file by applying the WinAPI

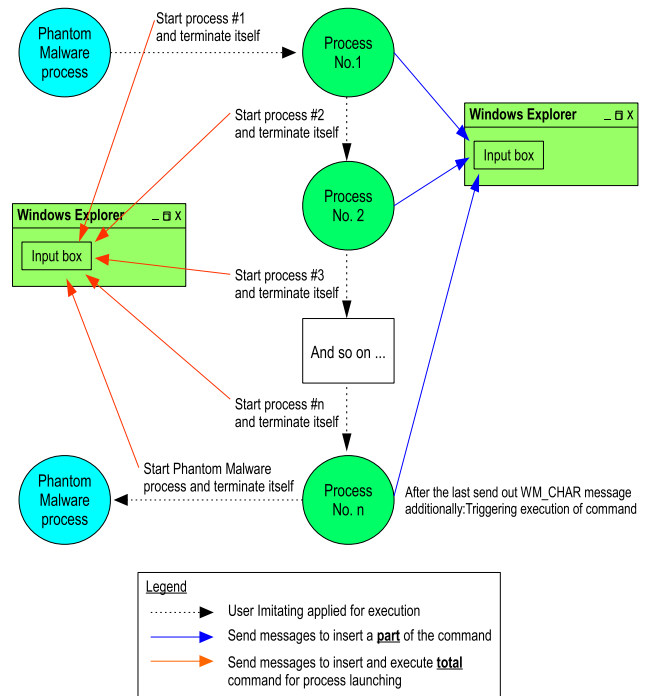


Figure 10. Conceal suspicious amount of sent Windows Messages by employing multiple processes with a masked parent/child process relation.

call CreateFileA [60]. In this file the encrypted command and its corresponding key for decryption will be written by the Phantom Malware using the WriteFile [65] WinAPI call. Afterwards, the Phantom Malware drops an exe file which inserts a part of the command if it is launched. The length of this part is random. User Imitating is not applied for the file dropping. Following, the Phantom Malware launches the exe file by applying User Imitating. The employed command to start this process has a length of approximately 20 characters. Finally, the Phantom Malware terminates itself (without employing User Imitating).

The launched process (started exe file) reads the stored encrypted command and key for its decryption, that belong to the file created by the Phantom Malware. The length of the inserted command part is determined randomly. The process sends messages to the WE’s input box in order to insert this part of the long command. Instead of inserting each character in its actual order, the characters will be inserted in a random order in combination with a corresponding movement of WE’s input box caret as mentioned in the previous subsection. Each character will be decrypted before sending it to this input box. Before decrypting the next character, the current character in memory must be overwritten. Before the process terminates, it starts another process which does the same actions. The length of the already inserted part will be passed over by the program argument while launching the next process (launching the exe file). To start this process, User Imitating is applied. However, the command for launching the next process must be inserted and confirmed for execution by the employed User Imitating technique in the input box

of another WE window. Otherwise, the parts of the long command, that were inserted beforehand, will be overwritten.

This insertion process continues until the long command has been inserted completely. Let n be the total number of processes which partly insert the long command in a sequence. The n^{th} process triggers the execution of the long command by sending a `WM_KEYDOWN` message [25] to the input box where this long command was inserted initially. Finally, the n^{th} process terminates itself and launches the Phantom Malware. User Imitating is employed for starting the Phantom Malware. To reach maximum obfuscation, the number of `WM_CHAR` messages [42] sent by each involved process must be randomized. Moreover, the number must be less than 20 due the assumed threshold of 40 `WM_CHAR` messages [42] which shall not be exceeded. As mentioned above, the child process sends out about 20 `WM_CHAR` messages [42] additionally in order to insert a command for starting the next process. In addition, to conceal a continuous sending of `WM_CHAR` messages [42] to the same text box, each process mixes randomized messages to other windows between its sent `WM_CHAR` message [42] sequence. The character insertion order of the two commands (command starting the next process and the long command) is rotated randomly. Characters of these two commands are inserted randomly alternating into the corresponding WE's input box.



For the operating system and the mentioned Phantom Malware detection technique it mistakenly looks like the involved processes are started by the user. This is due to User Imitating, employed for launching these processes. Hence, the parent/child process relation between these processes is masked. The command consists of about 2000 characters. Each process launched by the Phantom Malware inserts a part of the command which has a size of less than 20 characters into the WE's input box. In total, n is greater than 100. If a single process starts such an amount of processes by applying WinAPI calls, it would be seen as suspicious. In addition, the masked parent/child process relation between the involved processes for command insertion cause a separate counting of sent `WM_CHAR` messages [42] for each of these processes. The counter for `WM_CHAR` messages [42] sent by the Phantom Malware process will be reset. This is due to the process being terminated and started again after the long command has been inserted and confirmed for execution. As a result, the Phantom Malware process will not exceed the threshold for sent `WM_CHAR` messages [42]. This also applies in case of several command executions based on multiples processes that were completed in a sequence.

Assuming `WM_CHAR` messages [42] sent by a process and its launched child processes will be monitored by another Phantom Malware detection technique: The inserted command (in a text box capable for command execution such as a WE input box) is determined based on the monitored messages. The command is analyzed for malicious behavior. However, `WM_CHAR` messages [42] sent by the Phantom Malware process and its child processes will be monitored separately because of the masked parent/child

process relation. Each monitored sent `WM_CHAR` message [42] sequence of the involved processes represents only a part with about 20 characters of the total inserted command (approximate length: 2000 characters). The inserted command was obfuscated by employing DOSfuscation. As a result, the analysis of the command parts (that were split up) for malicious behavior will fail. Nevertheless, the inserted command in the other WE's input box for starting the next process will be determined completely. As mentioned above, the corresponding `exe` file was created without employing User Imitating but started by applying User Imitating. Hence, the operating system and Phantom Malware detection technique are fooled into perceiving that the user is launching a `exe` file created by a process.

To detect Phantom Malware, sent `WM_CHAR` messages [42] of all running processes must be monitored. Based on the monitored messages, the inserted command will be determined and scanned for malicious behavior. If the inserted command is malicious then the processes, which sent the `WM_CHAR` messages [42] for inserting this command, will be identified as Phantom Malware. However, monitoring the total amount of `WM_CHAR` messages [42] sent by all processes reduces the system performance. This stems from the fact that for message monitoring, a `WH_CALLWNDPROC` [66] hook is attached to all threads of every running process. This hook does not provide a filtering option [67] by default. In other words, each time a process sends out a message, the hook procedure will be called. Monitoring `WM_CHAR` messages [42] which are received by a window (containing a text box capable for command execution) will only detect an inserted malicious command. Processes started by the Phantom Malware which are responsible for this insertion will not be identified because messages do not have a tag for identifying its corresponding sender.

C. INCREASE OF EXECUTION SPEED

The Phantom Malware must execute multiple commands concurrently: As stated in the first subsection, for speed optimization purpose, the `&` operator [55] must be used to connect multiple commands in one line at the WE's input box until reaching the character capacity of this input box. The insertion and confirmation for execution of one command with an approximate length of 2000 characters by applying the multiple desktops-based User Imitating variant requires about three seconds. Upon opening the WE window and after the inserted command in the WE's input box is confirmed for execution, the input box's state is uneditable. Hence, a `WM_KEYDOWN` message [25] with  must be sent to this input box in order to set its state to editable. However, before this input box becomes editable, there is a "scrolling" animation by displaying listed directories. This animation takes approximately one second. Before inserting the command, the default text *Quick Access* (language specific) in the WE's input box must be removed by sending a `WM_KEYDOWN` messages [25] with  for each character of this text. In addition, the WinAPI call `SendMessage` [35] is applied

for inserting each command character. The calling thread is blocked by this WinAPI call until the receiver window has processed the corresponding `WM_CHAR` message [42]. The thread is blocked for about 1 ms. All in all, the command insertion needs about 2 seconds (2000 ms).

Assuming a single command (without any `&` operator [55] applied) has a length of about 25 characters, a total of 80 commands can be executed within more than three seconds. These commands are connected in one line at the WE's input box by using the `&` operator [55]. If there are thousands of commands to execute such as in the case of encrypting all victim's files, the total encryption process takes several minutes.

As mentioned in the previous subsection, the Phantom Malware starts one process which inserts a part of the command in WE's input box and which itself starts the next process before terminating. This next process does the same actions until the command is completely inserted. Instead of launching one process, the Phantom Malware starts multiple processes at the same time. In order to avoid race conditions, two WE windows are opened on the additional desktop for each of these processes by applying User Imitating. The command for starting the next process will be inserted and confirmed for execution in one of the WE windows. The actual command will be inserted in the other WE window by the process (launched by the Phantom Malware) and its child processes. After the actual command is completely inserted, the child process confirms it for execution. Finally, the child process terminates. However, only one child process of those which confirmed the commands for execution will launch the Phantom Malware before its termination.

V. EVALUATION

A. OPERATING SYSTEM COMPATIBILITY

As presented in table 1, the following operating systems are vulnerable for both User Imitating variants: Windows, OS X, Linux and Solaris. In contrast, Android and Qubes OS cannot be attacked by these variants.

The overlay-based variant has the following general requirements:

- Creating a full screen window (overlay window).
- Setting the overlay window on focus.
- Opening a window containing an input box where commands can be entered (in Windows: RDB and WE).
- Simulating keystrokes by sending messages to this input box (other process).

OS X, Linux and Solaris meet these requirements. The multiple desktops-based variant of User Imitating requires an additional desktop instead of a full screen window. OS X, Linux and Solaris do not have operating system specific API calls for additional desktop creation, like Windows does. However, these operating systems can be attacked by the multiple desktops-based variant by employing the overlay-based variant as a dropper technique:

- 1) Creating the overlay window.

Table 1. Compatibility of User Imitating variants with operating systems.

Name	Version	Overlay	Multiple Desktops
Windows	XP	✓	✓
	Vista	✓	✓
	7	✓	✓
	8	✓	✓
	8.1	✓	✓
	10	✓	✓
OS X	10.12 (Sierra)	✓	✓
	10.13 (High Sierra)	✓	✓
	10.14 (Mojave)	✓	✓
	10.15 (Catalina)	✓	✓
Linux	Ubuntu 19.04	✓	✓
	Debian 10.1	✓	✓
Solaris	11.4	✓	✓
Android	10	✗	✗
Qubes OS	4.0.1	✗	✗

- 2) Opening a window containing an input box where commands can be entered e.g. terminal.
- 3) Inserting the command into this input box for dropping the actual Phantom Malware (writing with the `echo` command [68] in combination with the redirection operator [50] bytes to an empty file for creating the executable), creating an additional desktop and starting the actual Phantom Malware there.
- 4) Closing the window.
- 5) Disposing the overlay window.

The execution of these commands by applying an API call such as `system` [69] would be seen as a suspicious action because the process launches an executable file which was dropped by itself.

For testing the compatibility of both User Imitating variants two programs were created for each operating system: Both test programs only move their own executable file located in the desktop directory to the documents directory for testing purposes. One program applies the overlay-based variant and the other one employs the multiple desktops-based variant for this action. Before both test programs are being launched on OS X from a terminal, a window appears indicating that these programs apply accessibility features (here: sending messages presenting keystrokes to the opened terminal window) as shown in Figure 11.

OS X blocks processes pushing Window Messages into the Event Queue of windows by default which do not belong to the window. Usually, only system/kernel processes have privileges for this action. In addition, the user/victim is allowed to define exceptions [70]. After the victim has defined an exception for both test programs by confirming this message, the programs will work flawlessly. In other words, both User Imitating variants will work perfectly. The victim can be

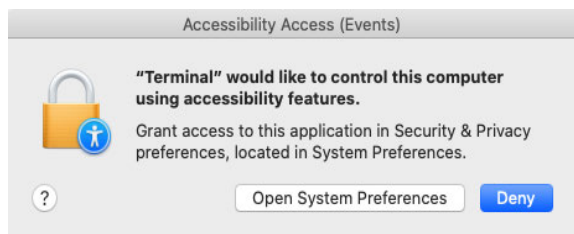


Figure 11. Message of OS X before launching a program employing User Imitating.

fooled into confirming this message by applying a social engineering technique [71].

Android does not meet the mentioned requirements of both User Imitating variants. By default, it does not provide a window with an input box where commands are entered e.g terminal. To make this operating system vulnerable for the overlay-based variant, a terminal application must be installed even before the actual attack. Moreover, instead of multiple desktops Android provides a multiple tab system: Only one window can be open at the same time. Every window is displayed in full screen. A tab represents a full screen window. After closing a tab of an application, another tab will be displayed. The user is able to switch between these tabs [72]. Assuming a terminal application is installed, the multiple desktop-based variant must be modified to attack Android: Instead of opening the terminal on an additional desktop, it will be launched on a tab in the background hidden from the victim. Android provides a functionality to display all tabs and switching to a specific tab chosen by the user/victim. Therefore, the victim will see the terminal tab and hence also the sequentially inserted characters. As a result, the victim becomes suspicious because these actions were not committed by them. Furthermore, if the victim closes its current active tab, the recently opened tab will be displayed - this could be the terminal tab. However, it is not possible to control the next opening of a tab by API calls after the current active tab is closed, because the tabs belong to different applications/processes [72].

Qubes OS cannot be attacked by both User Imitating variants. This operating system applies a compartmentalization. In other words, each instantiation of the underlying operating system such as Linux Debian runs on a different Virtual Machine (VM). Each VM is completely isolated from other VMs [73]. As a consequence, it is not possible to open a window in other VMs containing an input box where commands are entered. Both test programs cannot propagate into other VMs. However, both User Imitating variants work on Qubes OS if only one VM is used.

B. TESTING ANTI-VIRUS SOFTWARE

The described Phantom Ransomware in section IV was implemented in C++ for Windows 10. In addition, a corresponding ransomware without User Imitating was developed. This ransomware employs the same actions as the Phantom Ransomware: All malicious actions of this

ransomware without User Imitating are executed by applying WinAPI calls instead. For instance, the Phantom Ransomware copies a file by inserting the command `cmd.exe /c copy [source file] [destination file]` `> C:/.../output.txt` [74] into the WE's input box. In contrast, the ransomware without User Imitating employs the WinAPI call `CopyFile` [75] for the same action. Overall, the ransomware iterates over each file of the file system (except system files) and executes the following actions by using the WinAPI:

- 1) Reading the file.
- 2) Determining the corresponding encrypted file content.
- 3) Overwriting the file with this content.

Besides, this ransomware was written in C++. A whole variety of anti-virus software by different manufacturers for Windows 10 has been tested in order to detect the ransomware without User Imitating and its corresponding Phantom Ransomware. All modules such as behavior blocker of anti-virus software were active. If there was a configurable security level of the modules, the maximum level has been selected. Furthermore, if there was a ransomware protection available, this security feature was activated. In addition, the default ransomware protection included in the Windows Defender was also activated. These tests were performed in July 2020: All tested anti-virus software was up to date and capable of detecting the ransomware without User Imitating. However, none of them was able to detect its corresponding Phantom Ransomware as illustrated in Table 2.

Overall, both ransomware programs did the same malicious actions but the Phantom Ransomware executed its actions by applying User Imitating. As a result, User Imitating prevented the Phantom Ransomware to be identified as malware.





C. LIMITATIONS

The overlay-based variant of User Imitating creates a top-most window with full screen size. The window contains a screenshot that was taken before its opening. Therefore, while the overlay is open, the total desktop screen looks like it is frozen. If windows on the desktop contain frequently changed graphical elements such as video boxes, the victim notices a frozen image of this video. Besides, they are able to hear the corresponding volume. However, the victim is convinced that the frozen image of the desktop is caused by a temporary fault (lag) of a system process which is responsible for window displaying. This is due to the overlay relaying every received user input (keystrokes and mouse events) to the actually focused window in the background. After the overlay window is closed, the victim recognizes that the input has, in fact, been processed by the focused window. For instance, the keystrokes are displayed in a textbox. Over and above, the overlay is open for about less than one second. Therefore, the attack is barely noticeable for the victim during office work. Word processing programs such as Microsoft Word and Open Office contain textboxes instead of frequently changed

Table 2. Anti-virus software attempting to detect ransomware without User Imitating and the corresponding Phantom Ransomware.

Name	Version	Ransomware protection available	Ransomware without User Imitating	Phantom Ransomware
AVG	20.5	✓	✓	✗
Avast	20.5.2415	✓	✓	✗
Avira	15.0.2006	✓	✓	✗
Bitdefender	24.0.26.137	✓	✓	✗
Eset	13.2.15.0	✓	✓	✗
G Data	25.5.7.26	✓	✓	✗
Kaspersky	20.0.14.1085	✓	✓	✗
McAfee	16.0.13	✓	✓	✗
Norton	22.20.1.69	✓	✓	✗
Symantec	14.3	✓	✓	✗
Trend Micro	16.0.1277	✓	✓	✗
Windows Defender	10.0.19041.264	✓	✓	✗

graphical elements. However, the caret of these textboxes are not flashing during the attack. This is a result of the frozen image of the desktop caused by the overlay containing a desktop screenshot in the foreground (screenshot was taken beforehand).

The opening of the overlay window will be ignored by Windows (versions: XP to 10) when another full screen application is open. The full screen application remains in the foreground even though the overlay window requests keyboard focus. Opening the RDB window by applying the keystroke combination  +  causes a black screen. After approximately three seconds the desktop appears and the full screen application is minimized. While the black screen is shown, the command will be inserted into the RDB's input box and confirmed for execution. The victim does not notice the command insertion including the CMD pop up because they only see the black screen. The same applies to the opening of the WE window by employing the keystroke combination  + . Assuming the RDB or WE window are opened by applying the WinAPI call `CreateProcessA` [47], the black screen, followed by a switch to the desktop as well as a minimization of the full screen application, will also occur in case of the command being inserted into the input box of the RDB or WE window.

The multiple desktops-based variant of User Imitating is not noticeable for the victim as all actions do not appear on their current desktop. The additional desktop (created by using only the WinAPI call `CreateDesktopA` [45]) will be not displayed in Windows's desktop manager because this desktop is not completely initialized. In other words, further processes must be launched on this desktop to register it for Windows's desktop manager such as `userinit` [14]. Furthermore, the victim is unable to switch to this additional desktop by employing the corresponding keyboard shortcut.

The inserted commands in the input boxes of the RDB and WE window will be saved. In case of the user employing the input boxes, the provided autocompletion feature of these boxes will suggest the inserted commands.

VI. ANALYSIS

A. FAILURE OF ANTI-VIRUS SOFTWARE

Anti-virus software failed to detect Phantom Ransomware because behavior blockers were unable to perceive the malicious actions. The Phantom Ransomware masked these actions as user activity by applying User Imitating.

Behavior blockers protocol the process actions in a flow graph database. The actions include API calls, access to the file system etc. A database contains a set of flow graphs. Each of these flow graphs represents a predefined malicious behavior pattern. Concurrently to the monitoring of process actions, a heuristic control flow graph matching algorithm is employed to estimate similarities between the already monitored process actions (control flow graph) and the flow graphs in the data base. Based on the resulting probability, the process will be classified as malware [76]. Alternatively, based on the monitored API call sequence, the process is classified as being malicious or benign by applying a machine learning algorithm. For instance, Naïve Bayes or a Support Vector Machine [77].

Employed malicious behavior patterns in form of flow graphs, predefined by behavior blockers, consist of about five to seven actions. However, patterns for matching the mimicked user actions in form of the malicious command insertion based on WinAPI calls by Phantom Ransomware are nearly impossible to define. The consequence of there being no clearly defined instructions and order for the insertion of command characters of a specific command is explained hereunder. The number and order of the commands, which are connected together in one line at the WE's input box by employing the `&` operator, is random. Commands such as `move` [53] require file paths for the arguments passed over. These file paths (such as the desktop path) vary for each machine. This is because the username is contained in the paths. The inserted command is obfuscated randomly by applying DOSfuscation [63]. Each obfuscated command has a different length and character sequence, when comparing it to the same unobfuscated command. For explanatory purposes, it will be assumed that the unobfuscated command consists of the following three characters XYZ. It is also assumed that the

random DOSfuscation is limited by adding a random amount of escape characters \backslash to random positions only. For example, the following two random obfuscated commands would be possible: $X\backslash Y Z$ and $X\backslash\backslash Y\backslash Z$.

As shown in Table 3, there are two different WinAPI call sequences for inserting the two randomly obfuscated commands in the WE's input box. As explained, the two obfuscated commands are based on the same unobfuscated command. Overall, there is a different WinAPI call sequence for each possible random obfuscation of a specific command.

Table 3. WinAPI call sequences for the insertion of the two random obfuscated commands in the WE's input box.

Command	WinAPI call sequence for the insertion
$X\backslash Y Z$	SendMessage(hwnd, WM_CHAR, 'X', NULL) SendMessage(hwnd, WM_CHAR, '\', NULL) SendMessage(hwnd, WM_CHAR, 'Y', NULL) SendMessage(hwnd, WM_CHAR, 'Z', NULL)
$X\backslash\backslash Y\backslash Z$	SendMessage(hwnd, WM_CHAR, 'X', NULL) SendMessage(hwnd, WM_CHAR, '\', NULL) SendMessage(hwnd, WM_CHAR, '\', NULL) SendMessage(hwnd, WM_CHAR, 'Y', NULL) SendMessage(hwnd, WM_CHAR, '\', NULL) SendMessage(hwnd, WM_CHAR, 'Z', NULL)


The command characters will not be inserted in its actual order (from left to right). The character insertion order is randomized in combination with a corresponding caret movement of the WE's input box. Furthermore, the suspicious sending out of thousands of WM_CHAR messages [42] for the command insertion is concealed by applying multiple processes. Each process sends out a randomized amount of about 40 WM_CHAR messages [42], which is less suspicious. Due to the application of User Imitating to start these processes, there is no parent/child process relation between the processes. As a result, behavior blockers are unable to perceive that each of these processes inserts only a part of the total malicious command. The continuous sending out of WM_CHAR messages [42] of a single process is concealed by sending randomly generated messages to other windows between each sent WM_CHAR message [42].

The definition of malicious behavior patterns based on sent WM_CHAR messages [42] will result in raising falsely positive malware classification. For instance, tools for physically incapacitated people (such as voice control interacting with the WE window) will be mistakenly identified as malware. The same applies to the definition of malicious behavior patterns based on additional desktop creation by employing the WinAPI call CreateDesktopA [45]. Tools for desktop managing will be erroneously detected as malicious software, too.

Detection techniques, that are based on monitoring the API call sequence in combination with a machine learning algorithm [77], will not detect Phantom Ransomware. Only the API call's function name is considered without its corresponding arguments. As a consequence, the detection technique perceives a sequence of SendMessage [35]

WinAPI calls. Nevertheless, this WinAPI call is benign because it does not damage the system. Phantom Malware splits up its malicious actions into a sequence of non-malicious SendMessage [35] WinAPI calls. Defining a suspicious/malicious amount of SendMessage [35] WinAPI calls is not practical for any process. This is due to the fact that Phantom Malware applies multiple processes for inserting the command. Each of the involved processes inserts a random amount of command characters. For each inserted command character, SendMessage [35] is called once.

Behavior blockers do not determine the inserted command by monitoring the total sequence of sent WM_CHAR messages [42]. In addition, there is no analysis of malicious actions on the inserted command.

By receiving a WM_KEYDOWN message [25] with  the WE process will execute the command that was inserted in its input box. Instead of the Phantom Ransomware executing this command itself, it manipulates the trusted system process to execute the command. Here, the system process is simply fooled into perceiving that the user/victim has entered this command and also confirmed it for execution. In other words, the behavior blockers mistakenly assumes that this command execution is legit, namely that the "user" itself has entered this command and confirmed it for execution. As an example, the Phantom Ransomware connects to its C&C-Server for receiving orders (as described in section IV). In this case, for the operating system and the anti-virus software it mistakenly looks like the user is writing and executing a script to only connect to a server and receive data from it. Hence, the executed malicious action (as inserted and confirmed command in the WE's input box) cannot be traced back to the Phantom Ransomware process.

B. IDENTIFYING PHANTOM MALWARE'S TARGETED ENVIRONMENT

An environment is a system configuration such as the operating system's version and system language. Environmental-targeted malware applies an environmental check logic to identify their targeted systems. If the malware is launched on its targeted environment, it exposes a malicious behavior. Otherwise, the malware does not do anything harmful. In most cases, it terminates [78]. The following section summarises the two techniques GoldenEye [78] and VECG [79] for detecting the targeted environment of a malware. Further, it is discussed why these techniques failed to identify Phantom Malware's targeted environment.

1) GOLDENEYE

GoldenEye analyzes the malware's environmental check logic on assembly code level to determine the malware's targeted running environment as explained hereunder. Based on each possible return value of WinAPI calls representing an environmental query e.g. OpenMutexW [80], virtual environment spaces are constructed dynamically. Inside these environment spaces the malware is forced by changed return values to run on different execution paths (branches).

For example, when the malware applies the WinAPI call `OpenMutexW` [80], the malware will be launched on two parallelly running environments: In the first environment the return value is 0 and in the other environment the return value is 1. After the launch, each corresponding basic block (from the start of the branch until a jump instruction) is checked on whether or not it contains any malicious functions or related instructions leading to an interaction with the environment. If that is the case, the current environment is selected as being targeted by the malware. If the basic block applies a termination function such as `exit` [81], the current environment is not selected as a target of the malware. Otherwise, the next (starting from this basic block) WinAPI call representing an environmental query will be determined. Thereafter, the complete analysis process starts again [78].

2) VIRTUAL ENVIRONMENT CONDITION GENERATOR

Instead of triggering the environmental conditions of the malware by forcing it to execute each of its branches, the Virtual Environment Condition Generator (VECG) provides these conditions for affecting malware's behavior. To identify the environmental conditions of the malware, it is launched multiple times: During each execution of the malware, its invoked WinAPI calls are searched for to find out which of them represents environmental queries. Based on these queries, currently not satisfied environmental conditions are determined. In the next run, VECG satisfies these conditions to collect new conditions. For instance, the malware checks the existence of a specific registry key by using the WinAPI call `RegQueryValueExW` [82]. Then, the malware is launched on an environment where this registry key exists. The invoked WinAPI calls are analyzed for determining the new environmental conditions. Conditions identified in previous malware launches remain satisfied. Due to satisfying the current environmental conditions, the malware is lead into a new branch. The complete analysis process ends if all environmental conditions are satisfied [79].

3) REASON FOR FAILURE

Similar to concealing its harmful actions, the Phantom Malware is able to apply User Imitating for covering its environmental check logic. In other words, a command representing this check logic will be inserted in the WE's input box and confirmed for execution. The output of this command is relayed to a file by employing the `<` operator [50]. The Phantom Malware analyses the content of this file representing the environmental information and checks if it is launched on its targeted environment. The analysis process is similar to a string-searching algorithm. Overall, the Phantom Malware applies only the following WinAPI calls for its environmental check logic: `SendMessage` [35] and `ReadFile` [52]. The environmental query is concealed in form of a sequence of these two WinAPI calls. Besides, `CreateDesktopA` [45], `SetThreadDesktop` [46] and `CreateProcessA` [47] are used for additional desktop creation and opening the WE

window on it. Throughout the entire process, the mentioned WinAPI calls do not query any information about the victim's environment. Therefore, GoldenEye [78] and VECG [79] are unable to identify the environment targeted by a Phantom Malware.

C. DIFFERENCES TO RELATED WORK

1) UI REDRESSING

First of all, the User Imitating overlay variant conceals the execution of malicious commands of the malware as mimicked user actions. Tabjacking is applied for fooling the user into an unconscious confirmation of a selected action. This action was selected by starting a corresponding app which contains a UI element: If the user presses this UI element (e.g. button) then the action is executed, such as buying unwanted software in the Google Play Store. However, there is a restriction as to which actions can be executed by the tabjacking attack, as there must exist a UI element on an already installed app which executes the action by clicking on it. [10]. User Imitating is able to execute every action in form of a terminal command. The banking trojan only steals login credentials [15]. The UI is temporarily modified during these three attacks. Each attack modifies the UI in a different way which is presented in the following.

Any windows, that were opened by simulated user actions, are hidden by the overlay of the User Imitating technique. This overlay is of full screen size and always in focus. The tabjacking attack creates a full screen view in the foreground to cover the app that was opened in the background [8]. The overlay of the banking trojan is only placed on top of a specific app where the user must login (such as a banking app for example) [19]. The user shall be fooled into entering their login data in this overlay instead of in the underlying app.

The overlay of User Imitating contains a screenshot that was taken before the overlay was opened. The view created by the tabjacking attack contains a UI element at the same relative position as the UI element of the app that is open in the background. In the background app a specific action is selected. By clicking on the UI element of this app, the action is executed [10]. The overlay applied by the banking trojan is a copy of the underlying app. In other words, the overlay view and the underlying app view have the same appearance. The overlay contains UI elements which are indistinguishable from the UI elements of the underlying app [17].

Due to the screenshot in the overlay that was created by the User Imitating technique, the victim is fooled into thinking that this screenshot is the actual desktop screen. They only see the full screen sized overlay (desktop screenshot) in the foreground. The appearance of the view (based on UI elements, animations etc.) in the foreground, created by tabjacking attack, shall trick the user into pressing the UI element contained in this view [8]. This UI element has the same relative position as the UI element of the app opened in the background. The overlay of the banking trojan cannot be

differentiated from the underlying app. As a result, the user mistakenly perceives the overlay to be the original app [19].

The user tries to interact with the desktop presented by the overlay of User Imitating. As the user cannot use the UI components presented in this screenshot (such as buttons and text fields - the overlay only contains a picture), the user does not notice any reactions to their input. Furthermore, this screenshot is a still image. For instance, the user notices a frozen video on a website but they are able to hear the corresponding volume. Hence, the user becomes suspicious. The user is able to interact with the overlay of the banking trojan and the foreground view of the tabjacking attack because both views contain UI elements [8], [18]. As such, the user perceives a reaction to their input such as an inserted character in a text field induced by a key event. Besides, dynamic UI elements such as embedded animations do not appear as a frozen image.

Every user input (keystroke and mouse events) is captured by the overlay of User Imitating as the overlay is of full screen size and placed in the foreground. The input will be relayed to the actually focused window (which was on focus before the overlay opened) in the background. While the overlay is open, a window will be opened containing a text field where commands can be entered. As a result, the command will be inserted in there. The overlay hides the opened window and prevents a disturbance by the user. While the command is inserted, the user is unable to insert additional characters in the text box of this window. The view created by tabjacking attack in the foreground is transparent for every UI event. In other words, every occurring UI event on this view is automatically relayed by the operating system (Android) to the view of the open background app. The appearance of the view in the foreground shall fool the user into clicking on the UI element which has the same relative position as the UI element of the app in the background. If the user clicks on the UI element in the foreground, an `OnClick` event [11] occurs on the UI element in the background. The selected action, such as buying an (unwanted) app, will then be executed [8]. For malware detection software (anti virus) it mistakenly looks like the execution of this action has a legit basis meaning that the execution was intended by the user. The same applies to the User Imitating technique. However, the user is able to disturb the tabjacking attack because the total view in the foreground is transparent for UI events. For instance, the user can press on the view itself instead of clicking on a UI element. At the same position in the background app there is another UI element. By clicking on it, the selected action is changed. The banking trojan relays the characters, that were supposed to be entered in a text field of the overlay, to the corresponding text field of the underlying app. Due to the overlay, the user is not able to insert characters into the text fields of the underlying app. Meanwhile, the entered user input is being monitored [16]. The User Imitating technique does not record this input.

As mentioned above, User Imitating inserts the command into the text field where commands can be entered. The

character insertion is induced by sending messages to this text field that represent key events. As a consequence, User Imitating is independent from any user interaction. In other words, the user does not need to be fooled into clicking on a UI element of the overlay to trigger the execution of the inserted command. For the tabjacking attack, however, this user interaction is required [8]. The banking trojan is dependent on user interaction: If the user does not enter their login data in the overlay then the attacker is not able to receive the credentials [19].

The overlay of User Imitating is always in focus: If it loses focus it will request focus again. As a consequence, the user is unable to switch to other windows or see the opened windows (e.g. RDB and CMD) and the corresponding icons in the taskbar. The tabjacking attack and the banking trojan only place a view over another app [8], [17]. This view is not allowed to be in focus at all times. Otherwise, the user would be unable to switch to another view or app. However, Android does not provide a focus request for apps [72] so, the user is able to switch to other apps. They notice the app in the background that was opened by the tabjacking attack. In case of the banking trojan, the user sees a duplicated view where they can login. In both cases these views were not created by the user themselves. Therefore, the user becomes suspicious. Switching to other windows is prevented by User Imitating (on Windows) because the overlay constantly requests focus. Moreover, icons of open windows are displayed in the taskbar, although the taskbar itself is overlaid with a screenshot before the windows have been opened. Therefore, the victim will not see these icons in the taskbar (presented in the screenshot).

The overlay applied by User Imitating is created even before the window, containing a text field that is able to execute an inserted command, is opened. This masks the opening (pop up) of the window in combination with the overlay having the *always on top* property. The tabjacking attack creates the view in the foreground before opening the app in the background [8]. As a consequence, the user only perceives the view in the foreground instead of the app in the background. The banking trojan on the other hand creates the overlay after the specific app has been opened [19].

The overlay of User Imitating closes after the command has been executed. Afterwards, the actually focused window is set back to focus. Over and above, the overlay is open for about 500 ms. Due to the user input being relayed to the actually focused window, the user mistakenly perceives the frozen image (caused by the overlay) as a temporary fault of a process for desktop displaying. The view created by the tabjacking attack in the foreground is not closed after the user clicks on the UI element: The view is closed by the user themselves [8]. Once the user clicks on the UI element in the foreground, the app in the background will be closed. The overlay of the banking trojan closes if the user confirms their entered login data, such as by pressing a button. Thereafter, the recorded login credentials are sent to the attacker. The event for confirming the login data is mimicked to the

app which was underlaid by the overlay. This results in the app processing the login data [17], [19].

In summary, the overlay variant of User Imitating has the following advantages over the related work:

- Every action can be executed by this attack in form of a terminal command. There does not exist a restriction as to which actions can be executed.
- The user cannot disturb the attack by interacting with the total desktop/screen.
- The attack does not require an interaction with the user. The user does not need to be fooled into clicking on a UI element to trigger the execution of an action.
- The user is unable to notice windows opened by this attack (such as RDB and CMD).

In comparison with the related work, the only disadvantage of the overlay variant of User Imitating is the frozen image induced by the presence of the overlay itself.

Moreover, the multiple desktops variant of User Imitating covers the open window where the command is inserted by opening it on an additional desktop. This desktop is an additional instance for user interaction. In other words, the current user and the User Imitating multiple desktops variant do not share the same user interface. As a consequence, this variant does not need an overlay to cover the open window where the command is inserted. Overall, the multiple desktops variant of User Imitating has the same advantages and no disadvantages as the overlay-based variant, compared with the related work.

2) BADUSB DEVICES BASED ON KEYSTROKE INJECTION

Phantom Malware is a type of malicious software. A BadUSB device is a USB device simulating a further hidden (not obviously visible) USB device with a malignant behavior [20]. Hence, the BadUSB attack requires physical access. Therefore, it must be plugged into the victim's machine. The installation of the Phantom Malware is the same as the malware installation. In many cases, the victim is fooled into opening an infected file by a social engineering technique. Here, physical access is not required.

BadUSB devices are applied to drop and install malware on the victim's machine by mimicking keystrokes [28]. The Phantom Malware applies User Imitating to conceal the execution of all its malicious actions. Messages representing key events will be sent to a textbox by the Phantom Malware. These messages will insert and confirm the malicious command in the textbox, which itself is capable of executing the command. The simulated key events by User Imitating are software-based. Hence, the Phantom Malware is able to emulate key events to windows which are not in focus by sending corresponding messages to them. The keystrokes mimicked by a BadUSB device are hardware-based. During the installation of a BadUSB device, the operating system will mistakenly perceive it as being a "keyboard" because of its modified firmware. This "keyboard" responds to key event pooling requests by the USB controller with corresponding

key events. These key events shall be mimicked [22]. The operating system (here: Windows) is tricked into generating internal messages such as `WM_KEYDOWN` [25] which will be sent to the focused window [14]. As a consequence, the BadUSB device is only capable of simulating keystrokes for the focused window.

The BadUSB device opens a terminal by emulating a keyboard shortcut, which is specific to the operating system. Because this terminal window has focus, it is in the desktop foreground. As a result, the victim notices the open window containing characters that were or are inserted into its terminal line. These characters were or are not typed by the victim themselves. This results in the victim becoming suspicious. Also, this attack can also be disturbed by the victim. This would be the case when both the victim and the BadUSB device type concurrently or if the victim sets another window to focus. The User Imitating technique, applied by the Phantom Malware, conceals the existence and any additional pop up of open windows. The opening and pop up would be evoked by messages sent by the User Imitating technique. The concealing is based on opening the windows on a not active, additional desktop. This additional desktop does not have keyboard focus. Hence, a disturbance by the victim is prevented.

If the BadUSB device is plugged out by the user/victim, then the attack (malware dropping and installation) is stopped. In any case, the attack is finished in less than approximately one second. To stop the attack of a Phantom Malware its corresponding process must be identified and terminated.

Phantom Malware checks the successful execution of its command. The `>` operator [50] is used to redirect the output of the command into a file. Afterwards, the content of this file will be read and analyzed. Therefore, Phantom Malware is able to react in cases of an error. BadUSB devices cannot react in such cases as they are not able to detect them in the first place. The keyboard mimicked by the BadUSB device is only able to interact with the victim's system in form of responding to polling requests by the USB controller. As a result, this keyboard cannot interact with the file system such as reading files.

VII. FURTHER RESEARCH AND IMPROVEMENTS

A. ANTHROPOMORPHIC KEYSTROKE SIMULATION

`WM_CHAR` messages [42] sent to a window element are identical to key events for this element [27]. In case of a text box, a character corresponding to the key event will be inserted there. On the whole, Phantom Malware sends out approximately 1000 `WM_CHAR` messages [42] per second. the Phantom Malware waits until a message has been processed by the window. The applied WinAPI function `SendMessage` [35] blocks the calling thread until this has happened [35]. Overall, the thread is blocked for about 1 ms.

The inter-keystroke interval (IKI) is the time difference between two key press events [83]. The Phantom Malware

has an IKI of about 1 ms. In contrast, the human's average IKI amounts to approximately 140 ms [84]. For this reason, the character sequence inserted by the Phantom Malware is too high to be caused by a humane user. A human is unable to type with a writing speed of about 1000 characters per seconds. A next research step would be the development of techniques for giving the character insertion sequence by the Phantom Malware a more humane characteristic. For instance, the waiting time between two sent `WM_CHAR` messages [42] representing two adjoining keys on the keyboard must be lower than for keys that are far apart from each other. Also, the Phantom Malware increases its execution speed by inserting commands concurrently in input boxes of multiple opened WE windows. A humane user cannot type in two or more different windows concurrently. As a result, this speed optimization must be disabled to mimic an authentic user. On the whole, Phantom Malware's execution speed will be decreased by a factor of how many processes are launched for multiple command insertion.

B. AUTOMATION OF PHANTOM TRANSFORMATION

The conversion process of a malware without User Imitating into its corresponding Phantom Malware version is called Phantom Transformation. It is possible to convert every malware without User Imitating into its corresponding Phantom Malware version. If a malicious action cannot be realized based on terminal commands, a script will be dropped and executed by employing User Imitating. Alternatively, an executable file (.exe), that only performs this action, will be dropped and launched by applying User Imitating. The Phantom Transformation was done manually for the mentioned Phantom Ransomware in section V. However, the converting process was intricate and time consuming. A further research step would be the automation of this transformation: The source code of a malware without User Imitating shall be scanned for malicious actions. Subsequently, parts of this source code, representing the malicious actions, shall be replaced with code leading to the same results as the replaced one, but here User Imitating shall be employed. The commands (inserted into the WE's input box) must be automatically modified to redirect their output to a file. As file paths vary for each machine, a placeholder character must be contained in the path to this file. Instead of inserting the placeholder character, its representing file path shall be inserted which was determined before. Accordingly, these modified commands shall be encrypted. In other words, storing the encrypted command in program memory, such as in the form of an array data structure with corresponding decryption key, is automated. Before compiling, automatically generated source code shall be added for implementing the following Phantom Malware functionalities:

- Select a random number of commands which will be connected in one line at the WE's input box by using the `&` operator [55]. In addition, the order of these commands is randomized.

- Before inserting a command, it will be completely decrypted. Following, this command will be randomly obfuscated by applying DOSfuscation [63]. Before decrypting the next command for its obfuscation, the current decrypted command in memory must be encrypted.
- To insert the encrypted command into the WE's input box, one of its characters must be decrypted. Afterwards, the character is sent to the WE's input box and inserted there. The current decrypted command character must be overwritten, before decrypting and inserting the next character. This process continues until the command has been inserted completely.
- Randomize the command character insertion order in combination with a corresponding caret position movement.
- Increase the execution speed by inserting commands into input boxes of multiple opened WE windows concurrently.
- Conceal a suspicious amount of sent `WM_CHAR` messages [42] by employing multiple processes. These processes are opened by applying User Imitating.
- Analyze the content of the file (which the command output is redirected to) for an occurred error based on the executed command.


C. OBFUSCATION OF ENCRYPTED COMMANDS

In consequence of obfuscation purposes, a random number of commands shall be selected by the Phantom Malware which will be connected together in one line by applying the `&` operator [55]. Each of these commands is encrypted in memory. In order to randomly obfuscate each command by employing DOSfuscation, each command must be decrypted before the obfuscation process. After a command is obfuscated, it shall be encrypted. Before decrypting and obfuscating the next command, the current command in process memory must be overwritten. However, the existence of only one decrypted command in memory at any time can be used for a detection by memory scanners. Based on a decrypted command, signatures shall be created to detect Phantom Malware. As a result, a further research step would be the development of an advanced possibility for command obfuscation: The commands must not be decrypted for obfuscation. In other words, an encrypted command must be randomly obfuscated by applying DOSfuscation.

D. STEAL SENSITIVE INFORMATION

In addition to inserting and executing a command in the WE's input box, the Phantom Malware interacts with specific windows for stealing information such as login data in password managers. Reading files that contain this kind of information by using WinAPI calls will be seen as suspicious.

The Phantom Malware opens Firefox on the additional desktop. As the next step, the Phantom Malware opens the password manager of Firefox (version: 70.0.1) by inserting

about:logins into its search bar. The inserted text will be confirmed by sending a WM_KEYDOWN message [25] with . Firefox's password manager enables the user to copy their login data (username and password) to the clipboard. As a consequence, the Phantom Malware sends corresponding Windows Messages to the Firefox window in order to trigger this copying process. Thereafter, the Phantom Malware reads out the clipboard by using the WinAPI call GetClipboardData [85], representing the login data.

However, the described attack does not work for password managers which request the current user to enter their password before copying the login data to the clipboard. In other words, password managers such as the one of Google Chrome (version: 79.0.3945.117) verify that this action is induced by the real user. Firefox does not employ this verification process.

A next research step would be the development of artificial intelligence (AI) based techniques capable of scanning the taken window screenshot to receive sensitive information, such as emails: As an example, Thunderbird (email client) is opened on the additional desktop. Hereafter, a screenshot of the opened window will be taken. An AI-based algorithm analyzes this screenshot and lists up all emails. Subsequently, the AI interacts with Thunderbird by sending Windows Messages in order to open emails. For each opened email, a screenshot will be taken and scanned by the AI to convert it into strings such as the subject and the body of an email.





E. ADVANCED USER IMITATING

Users conveniently interact with windows directly that provide the corresponding functionality in order to execute their desired action. Another option would be to insert and confirm a command in the WE's input box which executes this action. In consequence of giving the Phantom Malware's command execution a more humane characteristic, further research steps would be the development of techniques for interacting with specific windows on the additional desktop.









For instance, a file shall be moved to another directory. As mentioned in section IV, moving the file is part of the file encryption process by Phantom Ransomware. The Phantom Malware interacts with the WE window to perform this action. The window provides access to the file system by listing all files of the currently selected directory and moving these files to another directory by *Copy and Paste*. Corresponding Windows Messages are sent by the Phantom Malware to the WE window in order to execute the following actions for employing this *Copy and Paste* functionality:



- 1) The absolute directory path of the file which shall be moved into another directory will be inserted into the WE's input box and confirmed. The WE window lists up all files of this selected directory by showing them in form of icons.
- 2) The icon representing the file, which shall be moved into another directory, will be selected by mimicking

corresponding  and  keystrokes for the WE window.

- 3) By simulating the key event  +  for the WE window, the selected file will be copied to the clipboard.
- 4) The absolute directory path of where the selected file shall be copied to (destination folder), will be inserted and confirmed in the WE's input box.
- 5) By emulating the key event  +  for the WE window, the file copied to the clipboard (step 3) will be inserted into the destination directory. This directory was selected in the previous step.

As another example, a file shall be overwritten by the Phantom Malware. The overwriting process shall be similar to how the humane user would conduct it: Instead of applying a command for this action, the user interacts with an editor program. The Phantom Malware sends corresponding Windows Messages to the WE window for executing the following actions in order to open the file. For this process it uses Notepad [86] (preinstalled editor in Windows):

- 1) The absolute file's directory path will be inserted and confirmed in the WE's input box. All files of the selected directory are listed and displayed in form of icons.
- 2) The icon representing the file will be selected by mimicking corresponding  and  keystrokes for the WE window.
- 3) The context menu for this file icon will be opened.
- 4) By emulating corresponding  and  keystrokes for the WE window, the entry *Open with* will be selected in this context menu.
- 5) For confirming the selected context menu entry, an  keystroke is mimicked for the WE window. Afterwards, a window opens containing a list of programs which are able to open the file. This file was selected in step 2.
- 6) In this window, the entry for opening the file with Notepad [86] will be selected by simulating corresponding  and  keystrokes for the WE window.
- 7) By emulating an  keystroke for the WE window, the selected entry will be confirmed in order to open the file with Notepad [86].

The text contained in Notepad's text box represents the file content. To change this text, the Phantom Malware sends a WM_SETTEXT message [43] to this text box. Afterwards, the Phantom Malware emulates the key event  +  for the Notepad's text box by sending corresponding WM_KEYDOWN [25] and WM_KEYUP [26] messages to the text box. As a consequence, the file will be overwritten with the content of Notepad's text box. Instead of overwriting the file, the Phantom Malware is able to read the content of the text box (representing the file content) by sending a WM_GETTEXT message [87] to the text box.

VIII. MEASURES AGAINST PHANTOM MALWARE

As presented in section V, tested anti-virus software was not able to detect the Phantom Ransomware. The implementation of this ransomware was discussed in section IV. Over and above, state of the art malware detection techniques are focused on the interaction of a program with the operating system's API (system calls) for malware classification. Phantom Malware, however, interacts with the user interface (desktop) for executing its malicious actions.

There are various countermeasures that can be applied against it:

- Only the text box shall receive and process WM_CHAR messages [42] in case the window containing this text box is on the current, active desktop. Otherwise, the corresponding process of this window must buffer these messages. If this window is set to keyboard focus, the corresponding characters based on the buffered WM_CHAR messages [42] will be inserted into its text box. Between the insertion of two characters, there is a waiting time of about 500 ms which enables the user to react.
- Alternatively, if a text box receives a WM_CHAR message [42], it shall be set to keyboard focus. In this case, the window containing the text box exists on an additional desktop which is not currently active. Hence, there must be a switch to this desktop. As another option, the window shall be transferred to the current active desktop. This will result in the user immediately noticing if a command is being inserted into the WE's input box.
- If a user process sends messages to, for example, a window that belongs to another process, it would also be possible that the operating system blocks this action by default. Besides, this action will be reported to the user such as in the form of a message box. The user is able to give the process privileges for sending messages to a windows associated to another process. So far, OS X is the only operating system which reports an action like this.
- Nevertheless, a complete disabling of WM_CHAR messages [42] sent to windows, which do not belong to the same process, is not practical. This goes back to the reason that tools such as voice control are employed by physically incapacitated people. In this way it may be seen as a discrimination. Besides, tools for remote maintenance apply this functionality, too [71].
- A process shall be not allowed to switch to another (additional) desktop by employing a corresponding API call or terminal command without the user's awareness. Therefore, the operating system must notify the user about this action such as in the form of displaying a message box. Then, the user is able to allow the process to switch to its desired desktop.
- Windows, Mac OS and Linux present the existence of an additional desktop only if there is an explicit request by the user such as opening the desktop manager. As a consequence, the user is unable to perceive the Phantom

Malware's actions on the additional desktop. However, if there is an additional desktop, then its existence and any opened windows on this desktop shall be shown on the current desktop. For instance, the desktop manager shall be integrated into the desktop taskbar such as in Solaris (version: 11.3): Each desktop is displayed in the form of a rectangle on the right end of the desktop taskbar. All open windows on a desktop are displayed by this desktop manager as further rectangles inside the initial rectangle which represents the desktop. As a result, the user is able to notice any windows whose opening has not been caused by themselves and becomes suspicious.

- If user's actions induce a disabling of the operating system's security features such as the firewall or shadow files, then the user must identify themselves. This verification process shall be similar to Google Chrome's password manager. In this example, a window opens and the user must enter their password before shadow files are deleted. This also applies when the user has privileges for the action. Hence, it is secured that the real user has intended this action.
- User input (key and mouse events) is represented by the operating system (here: Windows) in the form of messages such as WM_CHAR [42], WM_KEYDOWN [25] and WM_LBUTTONDOWN [34]. These messages are sent to corresponding windows. Afterwards, the thread of the window will process the received message (thread message loop) [27]. However, the operating system does not differentiate between *real* user input and *simulated* user input. *Real* user input is induced by a device (e.g. keyboard: key events) and *simulated* user input is generated by sending a corresponding message to a window which represents this input (e.g. key event: WM_CHAR [42] message). The device driver will determine the exact form of user input (e.g. keyboard: pressed key). Thereafter, a WM_CHAR [42] message is pushed to the System Message Queue. Accordingly, a system process will send this message to the focused window. This window is unable to distinguish if this message was generated by a device driver (*real* user input) or by a process (*simulated* user input). A window must be able to differentiate messages that represent *real* or *simulated* user input. If these messages represent *simulated* user input, they shall contain an unchangeable tag such as PID (process identifier) which identifies the sender process. *Simulated* user input must be ignored by windows of system processes such as the WE. In case of physically incapacitated people, system processes must only react to *simulated* user input by a specific process e.g. voice control.

IX. CONCLUSION

This paper contributed the approach of the User Imitating technique: Two variants of this technique were introduced. Harmful actions of the malware are concealed by simulated

user behavior. The mimicked user actions induce a change of the user interface such as opened windows. In order to prevent the user to become suspicious, all changes induced by the simulated user actions must be hidden, too.

The concept of the Phantom Malware has been contributed by this paper as well. This novel type of malware executes all of its malicious actions by applying the User Imitating multiple desktop variant. Overall, this malware masks all of its malicious actions by inserting corresponding commands into a text box that is able to execute them. A window containing this text box is opened on an additional desktop. However, this desktop is currently not active (not on keyboard focus). The victim only perceives windows on their current, active desktop. Overall, they do not notice the command insertion and confirmation on the additional desktop.

The WE is a preinstalled file manager and a fundamental part of the desktop environment of Windows [14], which makes this process trustworthy. Due to inserting and confirming a malicious command in the WE's input box, this trusted system process is manipulated to execute the command. Anti-virus software will not block actions of a trusted system because it is considered bad practice. Besides, the operating system including modules of anti-virus software, such as behavior blockers, are fooled into perceiving that the user themselves has done the command insertion and its confirmation. In other words, it mistakenly looks like the execution of the malicious command has a legit basis with the user having intended the execution.

State of the art behavior blockers were unable to detect Phantom Malware. The creation of behavior rules (patterns) for matching the manipulation of the trusted system process based on faked user actions (inserting the command) is impossible. The reason is that there is no exactly defined insertion order of the command characters. Before inserting it, the command will be obfuscated by applying DOSfuscation. This obfuscation is random. For example, a random amount of escape characters will be added in between the command characters. A detection based on faked user actions will falsely classify tools for physically incapacitated people e.g. voice control as Phantom Malware. These tools are based on an interaction with other windows for allowing physically incapacitated people a barrier-free computer usage. This interaction is induced by sending e.g. WM_CHAR messages [42] to text boxes for text/command insertion.

Multiple processes with a masked child/parent relation are applied by the Phantom Malware which insert a random part of the command into the WE's input box. Therefore, limiting the amount of WM_CHAR messages [42], which a process is allowed to send before further messages are blocked, does not prevent Phantom Malware. In addition, it is not possible to detect Phantom Malware by monitoring and analyzing the command inserted by a process and its corresponding child processes.

Over and above, Phantom Malware is a serious threat for both private and commercial users. The measures against it,

mentioned in section VIII, shall be realized and inspire the development of further ones.

CONFLICT OF INTEREST STATEMENT

The author declares that the research was conducted in the absence of every commercial or financial relationships that could be construed as a potential conflict of interest.

ACKNOWLEDGMENT

The author would like to thank Karsten Hahn for valuable feedback and discussions. He would also like to thank Christian Burmester for the spell check and valuable proofreading. He acknowledge support by Deutsche Forschungsgemeinschaft (DFG) and Open Access Publishing Fund of Osnabrück University.

References

- [1] M. Fadli Zolkipli and A. Jantan, "An approach for malware behavior identification and classification," in *Proc. 3rd Int. Conf. Comput. Res. Develop.*, vol. 1, Mar. 2011, pp. 191–194.
- [2] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Proc. Int. Conf. Broadband, Wireless Comput., Commun. Appl.*, Nov. 2010, pp. 297–300.
- [3] W. Liu, P. Ren, K. Liu, and H.-X. Duan, "Behavior-based malware analysis and detection," in *Proc. 1st Int. Workshop Complex. Data Mining*, Sep. 2011, pp. 39–42.
- [4] G. Cabau, M. Buhu, and C. P. Oprisa, "Malware classification based on dynamic behavior," in *Proc. 18th Int. Symp. Symbolic Numeric Algorithms Sci. Comput. (SYNASC)*, Sep. 2016, pp. 315–318.
- [5] J. A. Marpaung, M. Sain, and H. J. Lee, "Survey on malware evasion techniques: State of the art and challenges," in *Proc. 14th Int. Conf. Adv. Commun. Technol. (ICACT)*, Feb. 2012, pp. 744–749.
- [6] M. Niemietz, "Analysis of ui redressing attacks and countermeasures," Doctoral thesis, Ruhr-Universität Bochum, Universitätsbibliothek, Bochum, Germany, 2019.
- [7] L.-S. Huang, A. Moshchuk, H. Wang, S. Schechter, and C. Jackson, "Clickjacking: Attacks and defenses," in *Proc. 21st USENIX Conf. Secur. Symp.*, Vancouver, BC, Canada, Aug. 2012, p. 22.
- [8] M. Niemietz and J. Schwenk, "Ui redressing attacks on Android devices," in *Proc. Black Hat Abu Dhabi*, 2012. [Online]. Available: <https://pdfs.semanticscholar.org/b059/62d6d6510f274a31edfae2c8babe9acaac2.pdf>
- [9] A. SankaraNarayanan, "Clickjacking vulnerability and countermeasures," *Int. J. Appl. Inf. Syst.*, vol. 4, no. 7, pp. 7–10, Dec. 2012.
- [10] M. Niemietz and J. Schwenk, "Out of the dark: UI redressing and trustworthy events," in *Proc. 16th Int. Conf. (CANS)*, Hong Kong, Nov/Dec. 2018, pp. 229–249.
- [11] *Android: View.OnClickListener*. Accessed: Oct. 4, 2019. [Online]. Available: <https://developer.android.com/reference/android/view/View.OnClickListener>
- [12] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on Web in Android, iOS, and windows phone," in *Foundations and Practice of Security*, J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, A. Miri, and N. Tawbi, Eds. Berlin, Germany: Springer, 2013, pp. 227–243.
- [13] *Android: View*. Accessed: Oct. 4, 2019. [Online]. Available: <https://developer.android.com/reference/android/view/View>
- [14] P. Yosifovich, M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More*, 7th ed. Redmond, WA, USA: Microsoft Press, 2017.
- [15] L. Á. Tefanko, "Android banking malware sophisticated trojans vs. fake banking apps," ESET, Bratislava, Slovakia, Tech. Rep., 2019.
- [16] A. Kalysch, D. Bove, and T. Müller, "How Android's UI security is undermined by accessibility," in *Proc. 2nd Reversing Offensive-Oriented Trends Symp. ZZZ (ROOTS)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–10, doi: 10.1145/3289595.3289597.
- [17] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: From two permissions to complete control of the UI feedback loop," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 1041–1057.

- [18] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu, "Understanding and detecting overlay-based Android malware at market scales," in *Proc. 17th Annu. Int. Conf. Mobile Syst., Appl., Services*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 168–179, doi: [10.1145/3307334.3326094](https://doi.org/10.1145/3307334.3326094).
- [19] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. Halderman, Z. Mao, and A. Prakash, "Android UI deception revisited: Attacks and defenses," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, May 2017, pp. 41–59.
- [20] N. Nissim, R. Yahalom, and Y. Elovici, "USB-based attacks," *Comput. Secur.*, vol. 70, pp. 675–688, Sep. 2017.
- [21] K. Suzaki, Y. Hori, K. Kobara, and M. Mannan, "DeviceVeil: Robust authentication for individual USB devices using physical unclonable functions," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2019, pp. 302–314.
- [22] F. Griscoli, M. Pizzonia, and M. Sacchetti, "USBCheckIn: Preventing BadUSB attacks by forcing human-device interaction," in *Proc. 14th Annu. Conf. Privacy, Secur. Trust (PST)*, Dec. 2016, pp. 493–496.
- [23] *Universal Serial Bus Specification*, Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, Palo Alto, CA, USA, Apr. 2000.
- [24] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Hoboken, NJ, USA: Wiley, 2008.
- [25] *WM_KEYDOWN Message*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/inputdev/wm-keydown>
- [26] *WM_KEYUP Message*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/inputdev/wm-keyup>
- [27] *About Keyboard Input*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/inputdev/about-keyboard-input>
- [28] E. Karystinos, A. Andreatos, and C. Douligeris, "Spyduino: Arduino as a HID exploiting the BadUSB vulnerability," in *Proc. 15th Int. Conf. Distrib. Comput. Sensor Syst. (DCOSS)*, May 2019, pp. 279–283.
- [29] *Invoke-WebRequest*. Accessed: Jun. 1, 2020. [Online]. Available: <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invite-webrequest?view=powershell-7>
- [30] B. Stroustrup, *The C++ Programming Language*, 4th ed. Reading, MA, USA: Addison-Wesley, 2013.
- [31] *Market Share Statistics for Internet Technologies—Operating System Popularity*. Accessed: Feb. 29, 2020. [Online]. Available: <https://netmarketshare.com/operating-system-market-share.aspx>
- [32] *GetForegroundWindow Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getforegroundwindow>
- [33] *SetWindowPos Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowpos>
- [34] *WM_LBUTTONDOWN Message*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/inputdev/wm-lBUTTONDOWN>
- [35] *SendMessage Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendmessage>
- [36] *WM_Activateapp Message*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/inputdev/wm-lBUTTONDOWN>
- [37] *GetGuiThreadInfo Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getguithreadinfo>
- [38] *GuiThreadInfo Structure*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/ns-winuser-guithreadinfo>
- [39] *SetForegroundWindow Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setforegroundwindow>
- [40] *SendInput Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendinput>
- [41] *GetWindowTextA Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getwindowtexta>
- [42] *WM_CHAR Message*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/inputdev/wm-char>
- [43] *WM_SETTEXT Message*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/winmsg/wm-settext>
- [44] *PostMessageA Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-postmessagea>
- [45] *CreateDesktopA Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-createdesktopa>
- [46] *SetThreadDesktop Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setthreaddesktop>
- [47] *CreateProcessA Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa>
- [48] *Startupinfoa Structure*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/ns-processthreadsapi-startupinfoa>
- [49] *CloseDesktop Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-closedesktop>
- [50] *About Redirection*. Accessed: Oct. 4, 2019. [Online]. Available: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_redirection?view=powershell-6
- [51] *Attrib*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/attrib>
- [52] *ReadFile Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile>
- [53] *Move*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/move>
- [54] *ReadConsoleOutput Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/console/readconsoleoutput>
- [55] *Command Shell Overview*. Accessed: Sep. 10, 2019. [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb490954\(v=technet.10\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb490954(v=technet.10))
- [56] *Beast: Overcoming Limits of Traditional Malware Detection*, G Data CyberDefense, Bochum, Germany, 2019.
- [57] *BCDEdit Command-Line Options*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/bcdedit-command-line-options>
- [58] *Vssadmin Delete Shadows*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/vssadmin-delete-shadows>
- [59] *Volume Shadow Copy Service*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows-server/storage/file-server/volume-shadow-copy-service>
- [60] *CreateFileA Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea>
- [61] D. Gookin, *Advanced MS-DOS Batch File Programming*. New York, NY, USA: Windcrest, 1991. [Online]. Available: <https://books.google.de/books?id=NXUuAQAAIAAJ>
- [62] M. Russinovich, *TCPView v3.05*. Accessed: Oct. 29, 2020. [Online]. Available: <https://docs.microsoft.com/en-us/sysinternals/downloads/tcpview>
- [63] *Dosfuscation: Exploring the Depths of cmd.exe Obfuscation*, FireEye, Milpitas, CA, USA, 2018.
- [64] *EM_SETSEL Message*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/controls/em-setsel>
- [65] *WriteFile Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile>
- [66] *SetWindowsHookExA Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa>
- [67] *HOOKPROC Callback Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-hookproc?redirectedfrom=MSDN>
- [68] *Echo*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/echo>
- [69] *Library Functions—System*. Accessed: Oct. 4, 2019. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxbd00/system.htm

- [70] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing," in *Proc. 1st Int. Workshop Random Test. (RT)*, New York, NY, USA, 2006, pp. 46–54, doi: 10.1145/1145735.1145743.
- [71] T. Witte, "Mouse underlaying: Global key and mouse listener based on an almost invisible window with local listeners and sophisticated focus," *ICST Trans. Secur. Saf.*, vol. 5, no. 15, Oct. 2018, Art. no. 155740.
- [72] M. Song, H. Song, and X. Fu, "Methodology of user interfaces design based on Android," in *Proc. Int. Conf. Multimedia Technol.*, Jul. 2011, pp. 408–411.
- [73] J. Rutkowska and R. Wojtczuk, "Qubes os architecture," Invisible Things Lab, Berlin, Germany, Tech. Rep., 2010.
- [74] *Copy*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/copy>
- [75] *CopyFile Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-copyfile>
- [76] S. Cesare and Y. Xiang, "A fast flowgraph based classification system for packed and polymorphic malware on the endhost," in *Proc. 24th IEEE Int. Conf. Adv. Inf. Netw. Appl.*, Apr. 2010, pp. 721–728.
- [77] D. Uppal, R. Sinha, V. Mehra, and V. Jain, "Malware detection and classification based on extraction of API sequences," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, Sep. 2014, pp. 2337–2342.
- [78] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "Goldeneye: Efficiently and effectively unveiling malware's targeted environment," in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Cham, Switzerland: Springer, 2014, pp. 22–45.
- [79] M. Alaeiyan, S. Parsa, and M. Conti, "Analysis and classification of context-based malware behavior," *Comput. Commun.*, vol. 136, pp. 76–90, Feb. 2019. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0140366418300410>
- [80] *OpenMutexW Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw>
- [81] *Exit, _Exit, _Exit*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/exit-exit-exit?view=vs-2019>
- [82] *RegQueryValueExW Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regqueryvalueexw>
- [83] S. Pinet, C. Zielinski, S. Mathôt, S. Dufau, F.-X. Alario, and M. Longcamp, "Measuring sequences of keystrokes with jsPsych: Reliability of response times and interkeystroke intervals," *Behav. Res. Methods*, vol. 49, no. 3, pp. 1163–1176, Jun. 2017.
- [84] V. Dhakal, A. M. Feit, P. O. Kristensson, and A. Oulasvirta, "Observations on typing from 136 million keystrokes," in *Proc. CHI Conf. Hum. Factors Comput. Syst. (CHI)*, New York, NY, USA, 2018, pp. 646:1–646:12, doi: 10.1145/3173574.3174220.
- [85] *GetClipboardData Function*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getclipboarddata>
- [86] *Windows Notepad*. Accessed: Oct. 4, 2019. [Online]. Available: <https://www.microsoft.com/en-us/p/windows-notepad/9msmlrh6lzf3>
- [87] *WM_GETTEXT Message*. Accessed: Oct. 4, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/winmsg/wm-gettext>



TIM NIKLAS WITTE was born in Osnabrück, Germany, in 1998. He is currently pursuing the B.Sc. degree in computer science with Osnabrück University, Germany. He is also a Cyber Security Researcher and a White Hat Hacker. His main research interests include attacks against operating systems and networks.

• • •