How Hackers Use Binary Padding to Outsmart Sandboxes and Infiltrate Your Systems

By Ryan Robinson

Published: 2023-05-18 · Archived: 2026-04-05 18:38:59 UTC

What is binary padding? How can you detect against threats using junk data in various ways to evade defensive systems and sandboxes? Read on to learn more.

Binary padding is the process of adding extra or junk data to a portable executable (PE) file that, while not changing the behavior of the binary, changes certain characteristics that can help with either obfuscating relevant code or defeating sandboxing solutions and detections.

This technique is not novel. It has been employed in various forms for several years to achieve different effects, all of which are related to evading defense mechanisms. So why are we talking about it now? We have noticed recently a lot of phishing campaigns using binary padding while targeting victims, for example Emotet and QBot. (Skip down to read more about this "PufferPhishing" technique.)

This blog will take a look into the what, the why, and the who when it comes to binary padding as an evasive technique for cyberattacks.

Why Would Attackers Use Binary Padding?

Historically, malware was *small* in size. A lot of logic for malware can be programmed into a very small executable. A small size for malware was a requirement not long ago, as computers had slower internet and not a huge amount of RAM. Large binaries used to be limited to files like installers or games, as these contain a large amount of resources such as extra files, images and libraries in order to function. Over 10 years ago, if a file was very large, for example over 5 MB in size, it would have been an indicator of non-maliciousness.

These days not so much. As computers today have magnitudes more memory and disk space compared to twenty years ago, file size of a binary became less of an issue. This has allowed new compilers and programming languages to optimize differently without considering the final binary size. New programming languages such as Rust and Go default to statically link libraries to the final binary which helps improve performance, but results in a larger final binary. For example, a "Hello, World" Go binary has a size in the megabytes.

Binary padding comes with many benefits for malware authors, let's discuss a few of them and the pros and cons for each.

Binary Padding Can Defeat Sandboxes with Size Limits

One of the most simple reasons for binary padding is to simply make the file bigger! This attempts to avoid analysis from automated malware analysis tools, such as traditional sandboxes, since most have an upper limit on

the size of files being submitted. It also hinders analysis of the file as someone trying to submit the file might not be able to upload such a large file in reasonable time limits, since the upload requires network transfer to the sandbox. This simple but effective technique allows malware to avoid being analyzed by virtue of being *too big to fit in the front door*.

For security vendors, the solution is simple: raise the limit of file size. This arms race comes with added costs for security vendors, but also adds cost to the malware author, the larger they need to make their malware to usurp the limits also makes the malware much bulkier and harder to transport to the victim. (There is an easy method that malware authors are able to use to make their inflated files easier to transport that we will talk about later in this blog.)

Exceeding Sandbox Size Limits with Binary Padding	
Pros for the attacker	**Cons for the attacker**
– Simple to implement for the malware author – Effectively defeats most sandbox solutions – Multiple ways to achieve	– Easy for cybersecurity tools to detect – Malware with binary padding can be reversed – Hard to transport files – Does not defeat tools with no size limit

Using Binary Padding to Change the Hash

"*I'm not like other executables*." Binary padding can also be used to change the hash for a malware. This technique can be effective to avoid hash-based detections, but not systems with genetic-based detection. The technique is achieved by a malware creating a copy of itself and inserting random data into it, thus ensuring that a cryptographic hash, for example MD5 or SHA, of the file will be unique for each. It is easy to achieve as only one bit is needed to be changed to change the hash.

The reason why this works is due to a concept in cryptography called *diffusion*. Diffusion means that if one bit of an input into a cryptographic hashing function is changed, then much of the output (digest) is also changed. This means that similar inputs do not produce similar outputs, hiding any relationship between the digest and the plaintext. This has the effect of avoiding hash-based detections, as we can see it is trivial to change the hash. This is less of an effective technique as most modern malware detection solutions have moved beyond hash-based and use genetic, signature, or behavior-based detections.

An example with the string diffusion*:*

String	**MD5 Hash Digest**
Diffusion	c9fc4714730359980e89007f327c5b7d
Diffu**z**ion	2e24821ec7e23fe19d8ec494a801928a

Notice how changing one letter completely changes the above hash digest?

Binary Padding to Create Unique Hashes	
Pros for the attacker	**Cons for the attacker**
– Easy to achieve for the malware author – Defeats hash-based detections – Multiple ways to achieve	– Slightly antiquated, it does not defeat more modern malware detection methods such as signatures or genetic detection – Often requires the malware to create a copy of itself to modify, which can be suspicious behavior to detect in itself

An example of this technique being used is by the Orangeworm group with their Kwampirs backdoor; it achieves it by inserting a randomly generated string into the middle of the binary.

Binary Padding for Obfuscation

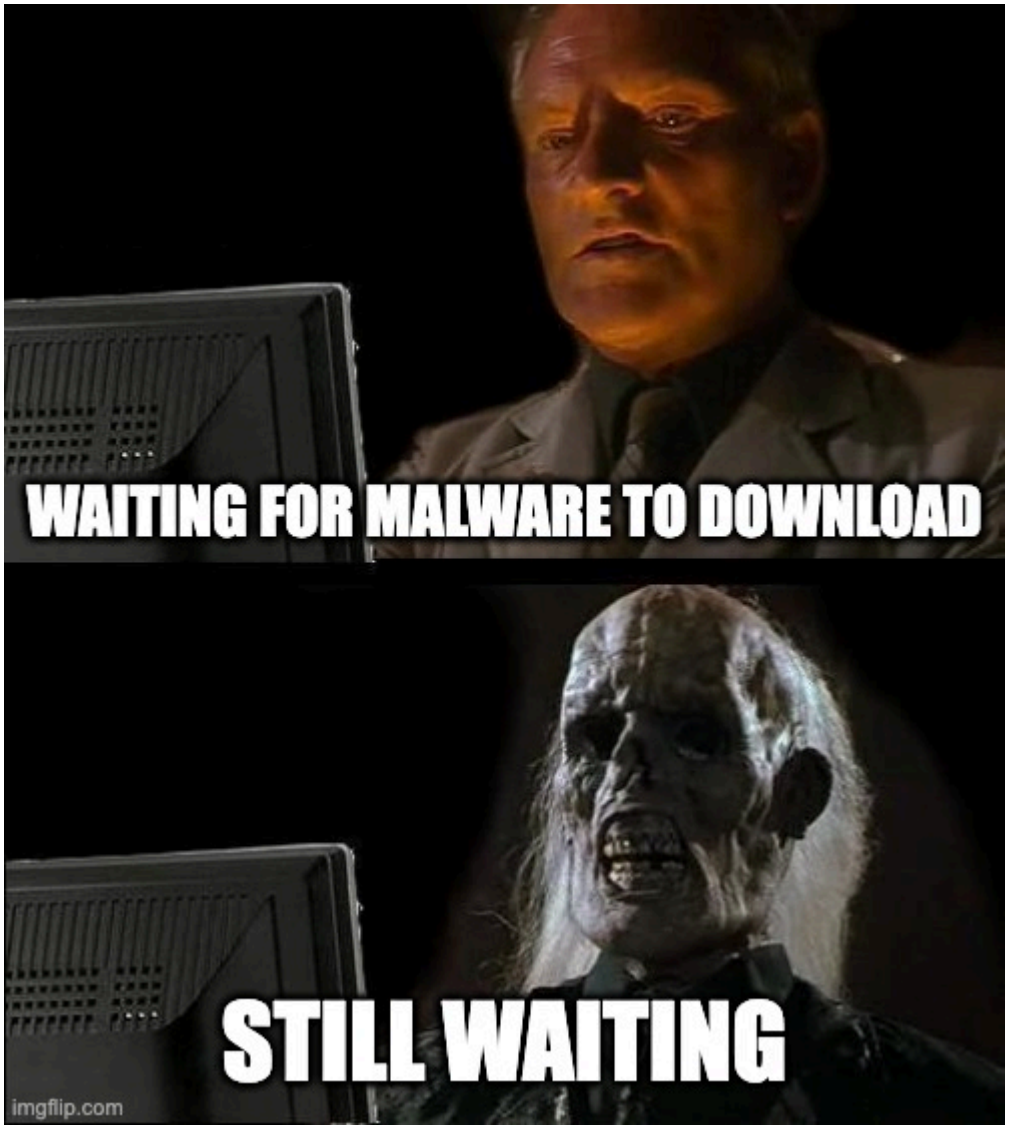
Padding, such as junk code, can be added to an executable solely for the purpose of creating so much junk and noise that it bogs down analysts when looking at the file, that it has the effect of obfuscating the relevant malicious code. Many would be familiar with the technique used in legislation of a *rider*; an extra provision added to a bill, hiding behind the main body of text hoping to fly under the radar in order not to be noticed. It is a similar concept with junk code obfuscation. Junk code has the effect of hindering manual analysis by a reverse engineer that does not have the tools or skills to avoid falling into the trap. While this is effective at hindering manual

analysis, it does not always have the ability to pass automated analysis tools as they can analyze code at volumes that cannot be performed by a human. The presence of junk code in itself can be a suspicious indicator of maliciousness; what can be less discreet than a malware containing the works of Shakespeare inside it?

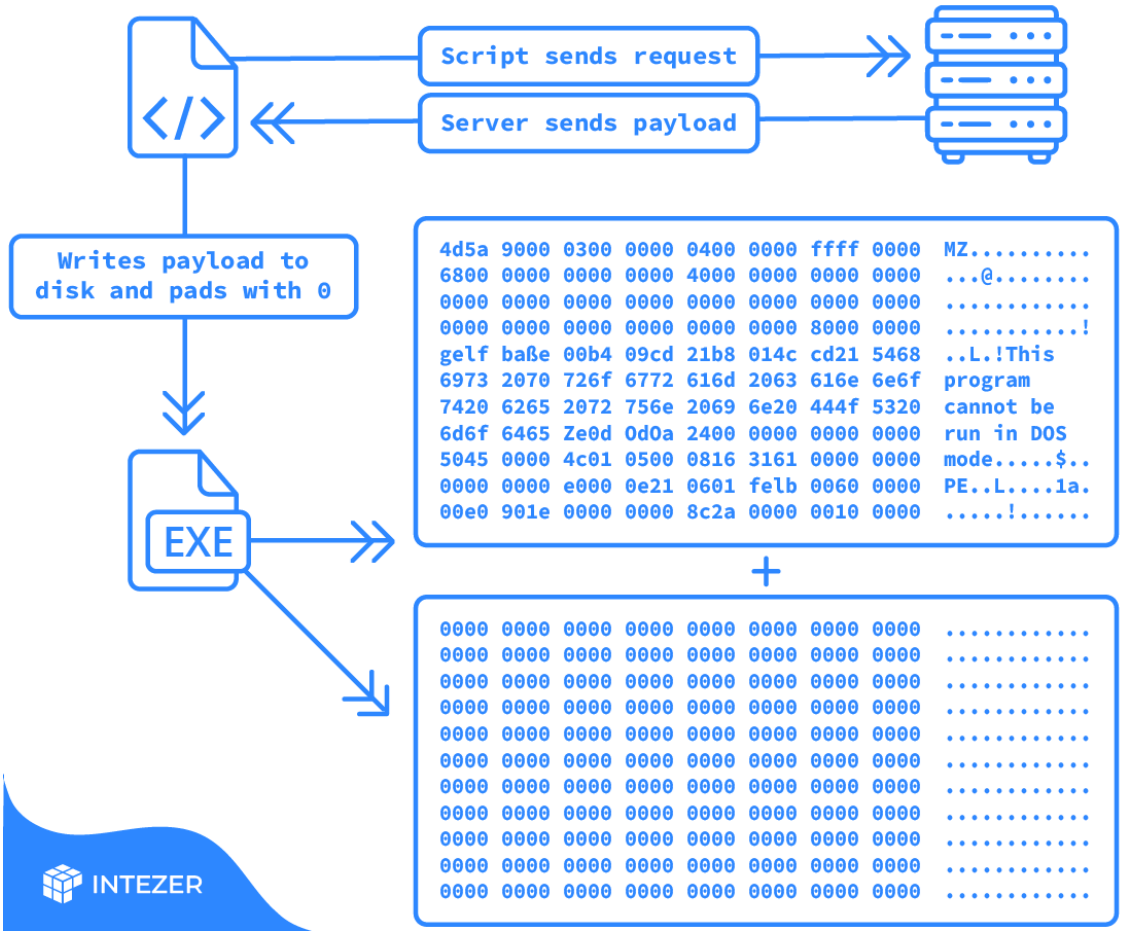
Binary Padding for Obfuscation	
Pros for the attacker	**Cons for the attacker**
– Can confuse malware analysts – Hinders debugging – Has some of the same positive effects of inflating the binary size – Can cause false negatives – Can cause decompilers and disassemblers to take a very long time to open the file	– Less likely to defeat automated analysis techniques – Can be a suspicious indicator in itself – Can cause more resource consumption, hindering the performance of the malware – Limited effectiveness on skilled analysts – Has the same negative effects of inflating the binary size

Problem for Threat Actors

The main problem for threat actors is that with binary padding, the increased file size can be hard to distribute. Threat actors want a file that is huge on disk, but does not take ages to deliver. Not everyone has an extremely fast internet, and therefore it would be inconvenient to a threat actor wishing to distribute malware to make it take hours to download.



What is their solution to this? The malware author can use a downloader that dynamically pads a binary after downloading it. An example of this is the threat group BRONZE BUTLER. BRONZE BUTLER targeted Japanese businesses with malware using padding. This was achieved by using a downloader that would download a payload and then insert the character '0' at the end of the file to inflate the size to 50-100 MB.

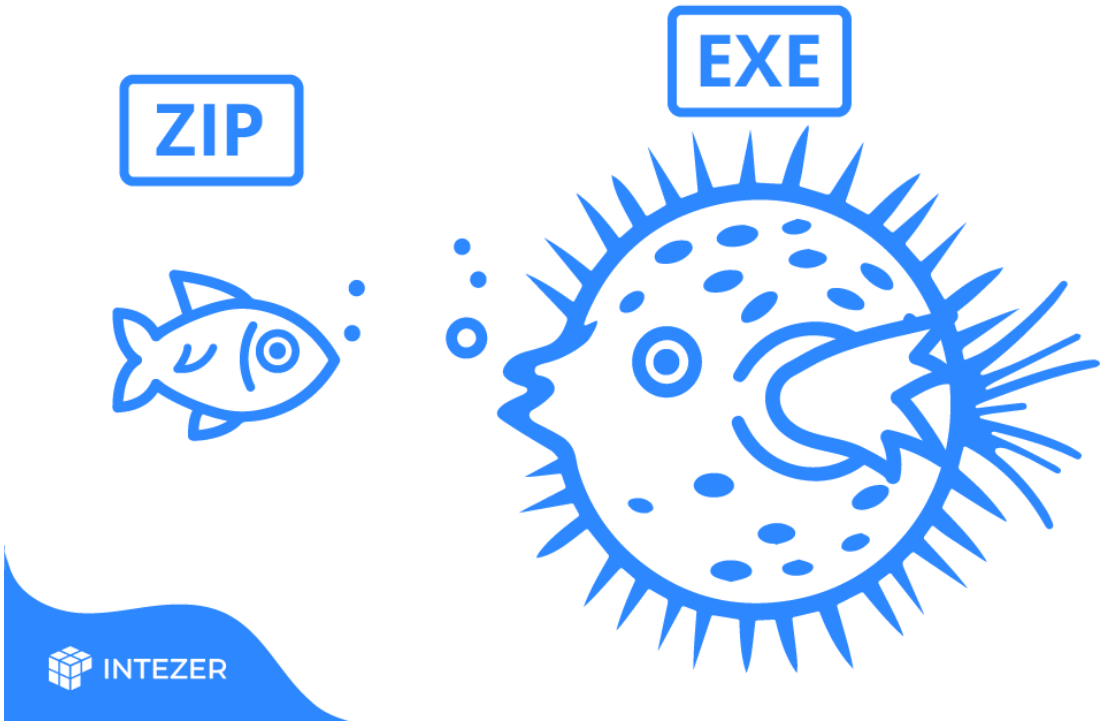


This comes with extra baggage that the malware author might not wish to have. Firstly the downloader itself might be detected, which can defeat the purpose of using binary padding to avoid detection. Next is also that it creates more network traffic that might be detected. The C2 might already be known to distribute malware, and it can be blocked or detected at this point. The use of a C2 can add an extra single point of failure in the chain where something might break. Also the payload that is fetched might be detected in memory or over the network before the padding is performed, so it might be analyzed before the padding has a chance to take effect.

There is another way to achieve padding without the same baggage that comes from using downloaders and C2 servers, **exploiting compression**. We will talk about this technique below, which we are calling "**PufferPhishing**" due to its active use in malicious phishing attachments. This is not a new technique, but is ever more being exploited by threat actors in phishing campaigns.

PufferPhishing: Binary Padding Combined with Compression

Recently we have noticed an uptick in the use of binary padding to evade analysis size limits… but with a smart technique to ensure that the files are still deliverable, by exploiting the efficiency of compression algorithms. Malware authors are creating padding that uses resources and overlays in order to create malware that is hundreds of MBs in size, but is able to compress to over 90% of its original bulk due to its efficiency in compression. This means that malware can be created to circumvent sandbox size limits but also solve the file transport problem that comes with that technique. It is the binary padding equivalent of having your cake and eating it too!



How Does the PufferPhishing Technique Work?

The technique works by using repeating sequences of bytes padded inside binaries to inflate the size of the binary. But the nature of the repeating bytes used allows it to be compressed to over 90% of its original size, commonly in the ZIP file format.

ZIP archive file format most commonly uses the DEFLATE compression format, incorporating LZ77 and Huffman coding. It is very good at compressing sequences of repeating data. DEFLATE compression works so well on repeating bytes that it's worth showing an example:

Creating a file of size 5 KB, with the repeating character `a` will compress to 22 bytes with DEFLATE. A 99.56% decrease in size. Whereas a file with the string below will only compress to 45 bytes:

```
aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
```

This is a 13.46% decrease in size. Through DEFLATE, the file with the repeating bytes will be smaller than the file with less repeating bytes even though the former file was around 100x bigger decompressed! This shows how the use of repeating data (or a very low entropy) can be exploited to make very large malware files that are easy to transport. The following CyberChef recipes have been created to help demonstrate:

File 1 (Repeating Bytes/Large File): Compression, Entropy

File 2 (More Entropy/Smaller File): Compression, Entropy

Compression likes lower entropy. Entropy is a measure of how random the data inside a file is. Since compression algorithms work by finding patterns to encode in a smaller format, a smaller entropy of data in a file will lead to

better compression ratios.

What does this look like incorporated into malware?

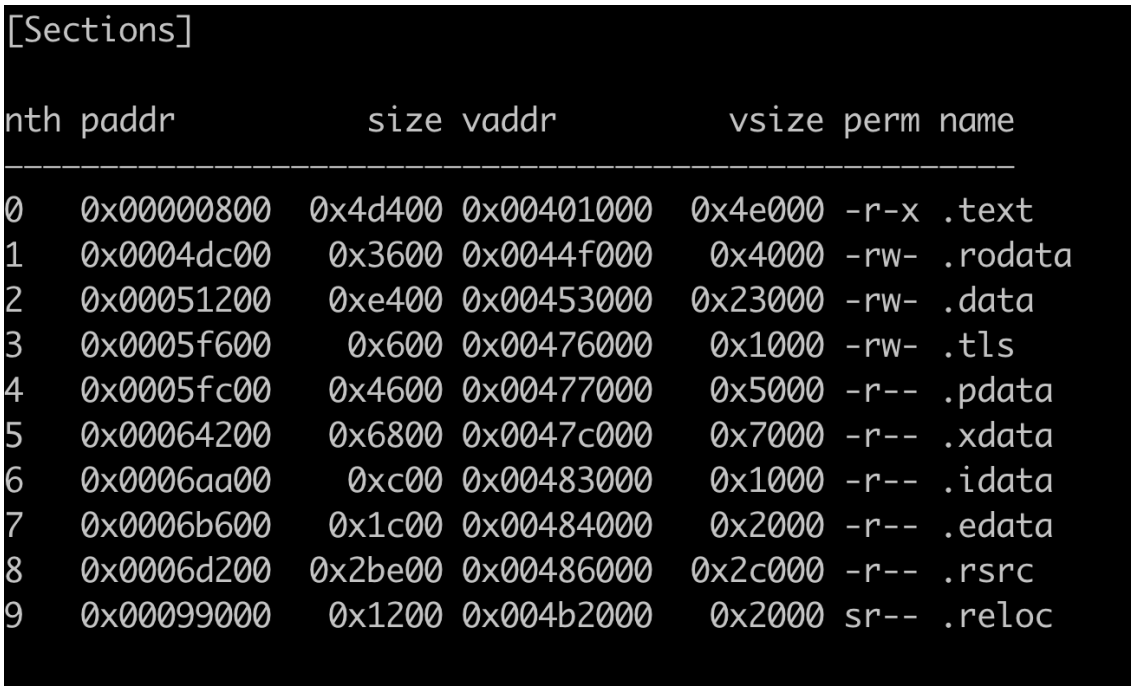
Examples of Malware with Binary Padding

Overlay

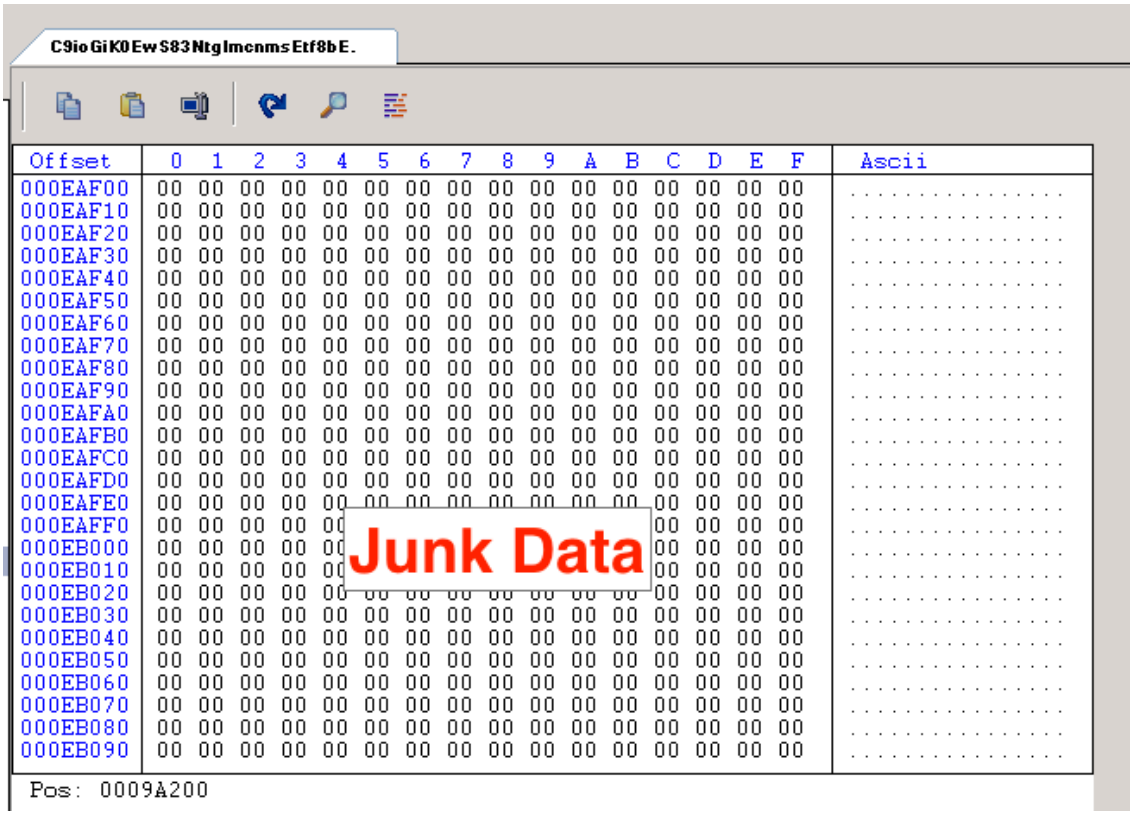
In March 2023, a Trend Micro blog noted Emotet using binary padding in the overlay in order to inflate the file size. Using 00-byte padding allowed the PE file to be compressed from a sizes of over 500 MB to ZIP files less than 1 MB:

61bb1c8adaaffc629e60db7e4a171538c5d625a6d79d225e03310c684efd7581

This file contains a very large DLL inside with padding in the overlay. The sections end at the file end at `0x9a200`.



Junk data is appended past the defined end of the PE. Most of it being the null byte.



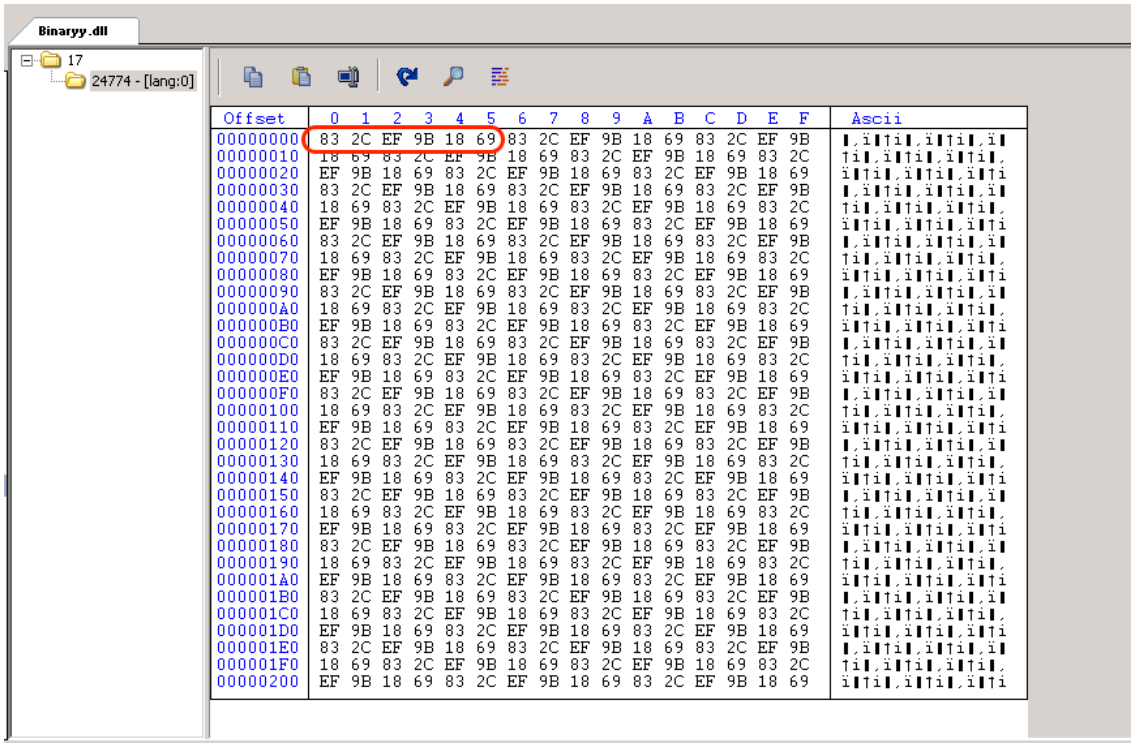
99.9% of the file is padding. The padding in the overlay is not loaded to memory when the PE is executed. This allows it to take up less space in memory than it does on disk.

Resources

The resource section of a PE is used for many items that are used as resources by the binary, this can be things such as images, audio, icons, and videos, but it can be abused. Very commonly the padding is placed in the resource section of malware. An example of this being used is in phishing emails, sending ZIP files that contain a Windows Installer (MSI) file:

34bcf0a864b6c7f4ecca92f1216ed58da4018c719552cfb62fee940924a91154

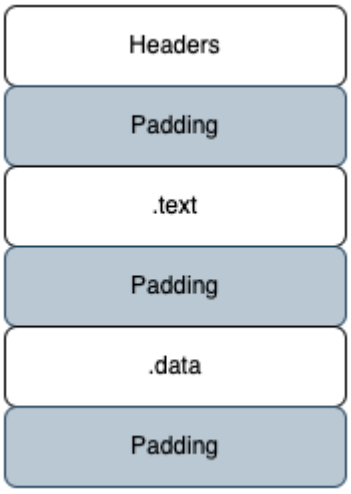
The MSI file is in Composite Document File V2 Document (CDF) format with an embedded DLL file that is padded using the resource section. The use of the CDF format also adds another layer of defense evasion that might bypass security detections that only work on PE files. The DLL is padded in the resource section with a stream of repeating bytes.



The MSI file is almost 500 MB but is able to be compressed to a size of around 2 MB, a huge reduction in size.

Space Between Sections

Similar to the overlay, malware authors can also play with the space between sections. Data can be inserted in between the sections of a PE to inflate the size.



This technique has been used by researchers in order to deceive malware classification.

Giant Variable

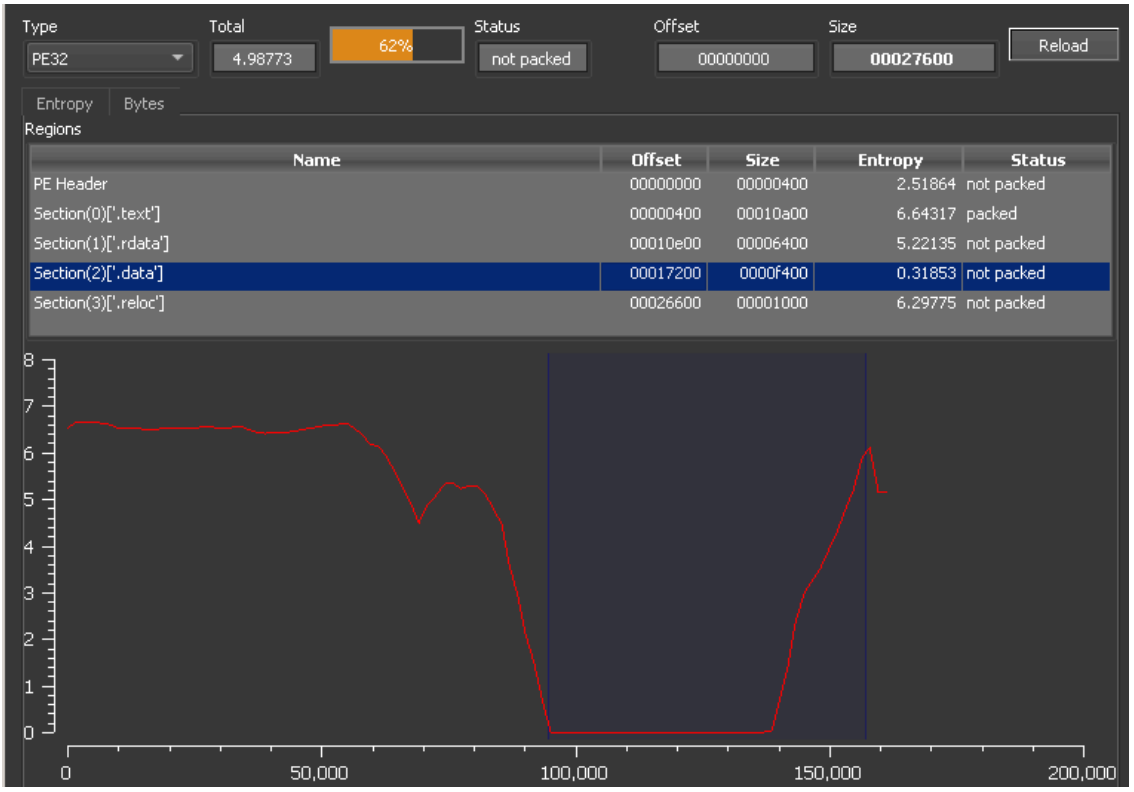
Malware authors can also use really large strings in order to pad malware in the data section. This technique is not often used as it can be easily detected, through looking for large strings. This technique involves simply creating

extremely large strings in the code of the malware that are not used specifically for the functionality of the malware itself. An example can be easily shown with a simple C program printing out very large strings.

```
int main()
{
    printf("0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    printf("1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    printf("2000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    printf("3000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    printf("4000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

    return 0;
}
```

These strings are stored in the data section of the PE. Since they are just sequences of repeating characters, they have very little entropy, and can help with compression. The graph below shows the drop in entropy of where these strings are stored. One issue that a threat actor might stumble into with this technique is that depending on the compiler, there can be a limit on the maximum length of a string literal. This could force the threat actor to use many smaller strings instead of one massive string for padding.

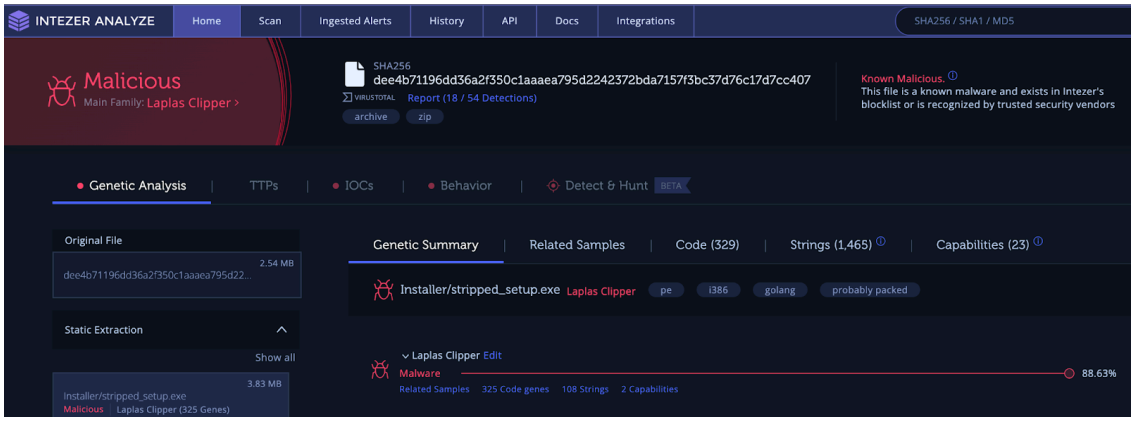


Compressing this file in ZIP format will lead to a 64% deflation size.

How Intezer Fights Binary Padding

Binary padding can be a simple and effective way to bypass security defenses, it is one technique out of many that are in a malware authors arsenal. This technique combined with other techniques, such as compression, can be chained to create innovative payloads that hinder analysis and detection.

Intezer is able to remove padding from samples to make it much more manageable to analyze. Take for example this Laplas Clipper sample. It is padded to over 600 MB through using the overlay technique! Intezer is able to unpack the sample from the ZIP file and remove the overlay. Give it a go yourself with padded samples. Submit the parent ZIP and Intezer will do the rest.



Source: https://intezer.com/blog/research/how-hackers-use-binary-padding-to-outsmart-sandboxes/