How IoT Botnets Evade Detection and Analysis

nozominetworks.com/blog/how-iot-botnets-evade-detection-and-analysis/

By March 1, 2022

On the Hunt for Elusive Malware

Malware families targeting the IoT sector share multiple similarities with their Windows-based counterparts, such as trying to evade detection by both researchers and security products. Their aim is to stay below the radar as long as possible. One key technique to stymie reverse engineering botnet code is to obfuscate the code by compressing or encrypting the executable, called packing.

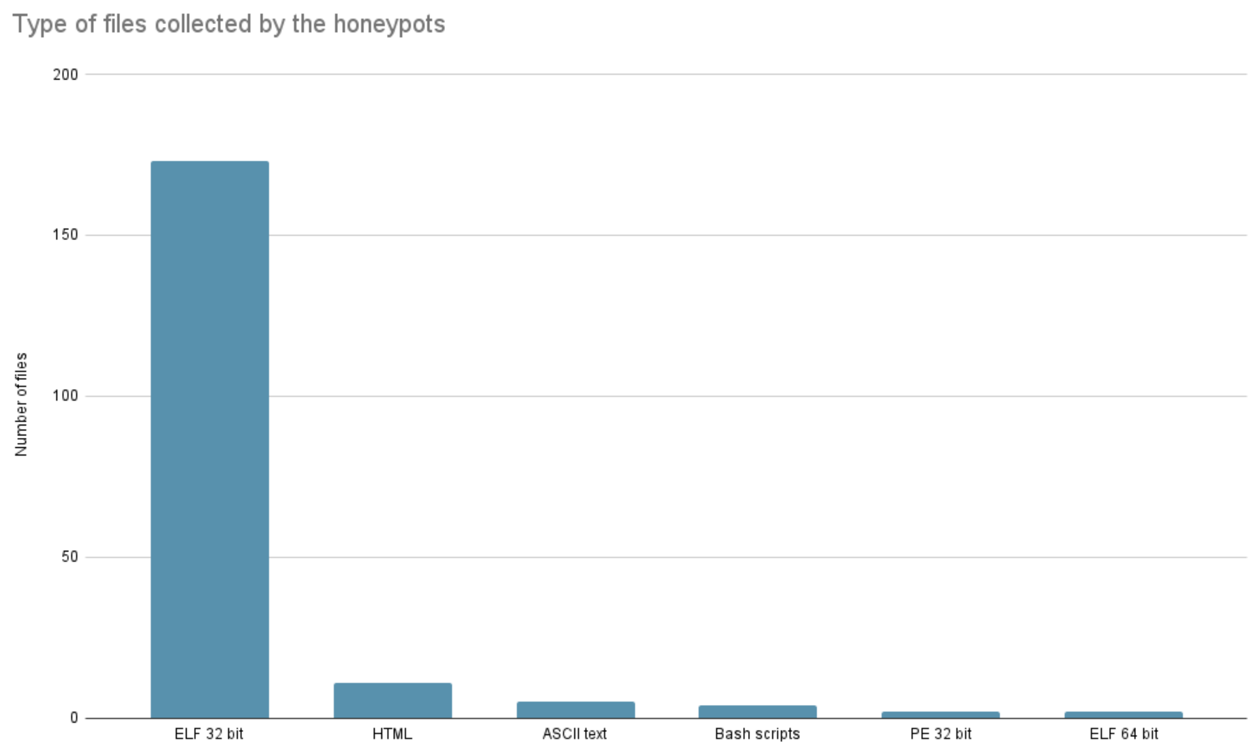
In this post, we explore the current status of the packers used by IoT malware, using data collected by Nozomi Networks honeypots. We'll also dig deeper into various obstacles that researchers face when analyzing obfuscated samples and show how they can be handled.

As malware targeting the IoT sector evolves, many features are being borrowed from the more well-known IT sector.

IoT Botnet Attack Statistics

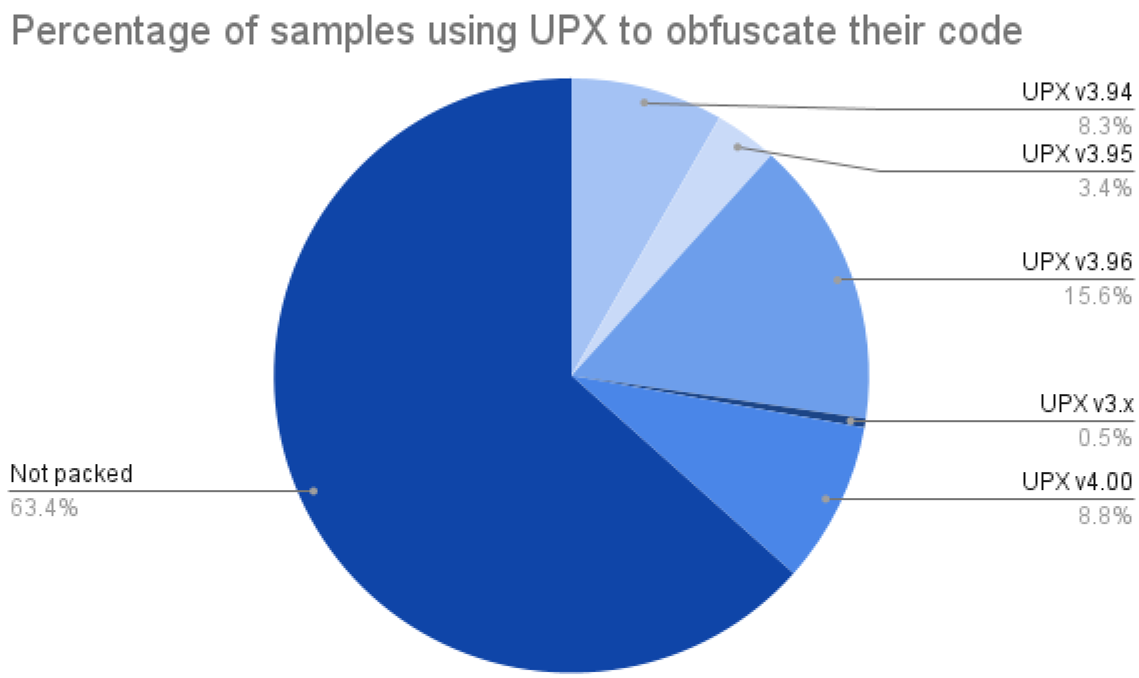
Honeypots are vulnerable systems that are deliberately made available to adversaries so that information on malicious activity can be collected for research, detection, and defensive purposes. Nozomi Networks Labs runs a network of passive honeypots across various regions that emulate multiple protocols commonly abused by attackers. The data they collect is aggregated and used for research and detection efforts.

Over the course of a seven-day period, Nozomi Networks honeypots collected 205 unique samples from attacks. The vast majority of the collected files are Linux executable files for 32-bit architectures.



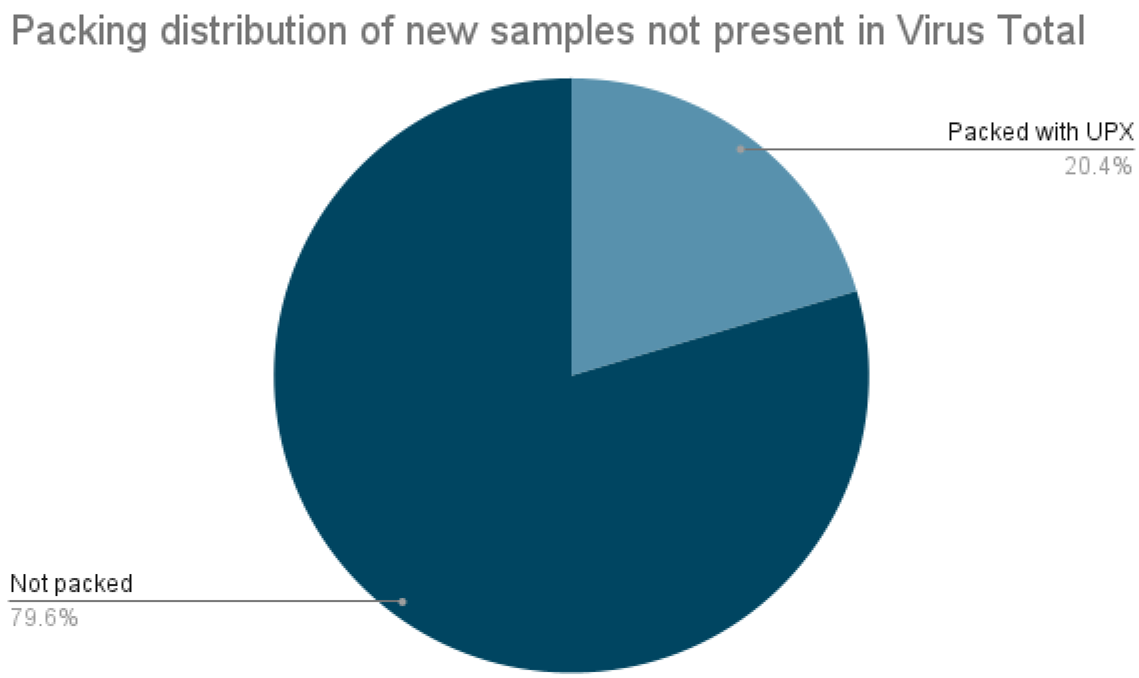
Statistics of the different file types collected by our honeypots.

Attackers use packers to obfuscate their code, concealing the original code with the intent of evading detection and making malware analysis more difficult and time consuming. Of the samples we collected, approximately one third were packed using multiple versions of UPX, a free and open-source packer widely used by both legitimate companies and malicious actors. UPX was the only packer used in this set of samples.



Percentage breakdown of UPX-packed samples and unpacked samples collected by the honeypots.

At the time of the analysis, 49 of 205 samples were not present on VirusTotal, a share site for malware details and research, and thus we decided to focus on these potentially new threats. This subset of files follows a similar percentage distribution regarding the packer and the version employed. Most of the new files—almost 80%—were not packed at all, with the remainder packed with UPX.
Based on initial research, one of these samples appears to belong to the same malware family that showed up in other internal Threat Intelligence research. It also stands out because of the particular UPX packing structure modifications.



Of new samples not present on VirusTotal, 80% were not packed, while the rest were packed with UPX.

Unpacking Challenges

When a sample is only packed with UPX, it is very easy to unpack with the standard UPX tool using the `-d` command line argument. Therefore, attackers will commonly modify the UPX structures of a packed sample so that it remains executable, but the standard UPX tool can no longer recognize and unpack it.

Since UPX is an open-source tool, we can check its source code on GitHub to understand its structures and what fields it uses. (You can see more of its file structure here.)

```
struct b_info {     // 12-byte header before each compressed block
    uint32_t sz_unc;            // uncompressed_size
    uint32_t sz_cpr;            // compressed_size
    unsigned char b_method;     // compression algorithm
    unsigned char b_ftid;       // filter id
    unsigned char b_cto8;       // filter parameter
    unsigned char b_unused;
};

struct l_info       // 12-byte trailer in header for loader (offset 116)
{
    uint32_t l_checksum;
    uint32_t l_magic;
    uint16_t l_lsize;
    uint8_t  l_version;
    uint8_t  l_format;
};

struct p_info       // 12-byte packed program header follows stub loader
{
    uint32_t p_progid;
    uint32_t p_filesize;
    uint32_t p_blocksize;
};

```

Sample UPX file structure.

Most IoT samples packed with UPX modify the `l_info` and `p_info` structs in the header. For example, as we have seen before with the SBIDIOT malware, it's common for malware authors to modify the `l_magic` value of the `l_info` struct in UPX-packed samples. In this case, unpacking the sample is as trivial as replacing the modified `l_magic` value with `UPX!` and executing `upx -d`.

In other cases, like the Mozi IoT malware family, the p_info struct is modified to set `p_filesize` and `p_blocksize` to zero. The solution then involves repairing the two zeroed values by replacing them with the filesize value that is available at the trailer of the binary.

However, when we tried to unpack the samples of interest, UPX returned an unusual error:

```
/samples/mw/7_2 > upx -t 0417.upx
                    Ultimate Packer for eXecutables
                       Copyright (C) 1996 - 2020
UPX git-d7ba31+ Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

testing 0417.upx
upx: 0417.upx: CantUnpackException: corrupt b_info

Tested 1 file: 0 ok, 1 error.
```

Malware sample fails while testing to determine if it can be unpacked with UPX.

In this case, UPX is telling us that there was a problem with the `b_info` structure. After some research, we concluded that this structure doesn't seem to be widely used by malware authors. `b_info` is a structure placed before each compressed block and contains information about its compressed and uncompressed size, along with the algorithm and parameters used to compress them. Once we checked the `b_info` structure in the file, we realized it hadn't been zeroed or modified in an obvious way.

Diving deeper into the internals of UPX, we found the exact place in the code that raises this exception. If the compressed size (`sz_cpr`) is bigger than the uncompressed size (`sz_unc`), UPX will fail. However, their values were coherent, so we discarded this modification as the source of the problem. In these lines of code, we can see that the most promising reason could be a problem with a difference between the sum of the declared size of the uncompressed sections and the declared size of the uncompressed file. In our sample, the sum of the value of `sz_unc` was bigger than the value of `p_filesize` , so we modified the appropriate `p_info` structure to set its `p_filesize` field with a value that wouldn't trigger this exception.

After changing these values, a header checksum exception was raised. Calculating this checksum value was possible since we had its source code, but it would be time-consuming, so we temporarily moved to another research path. We decided to create packed samples that were as similar as possible to the malicious sample so it would be easier to spot the differences.

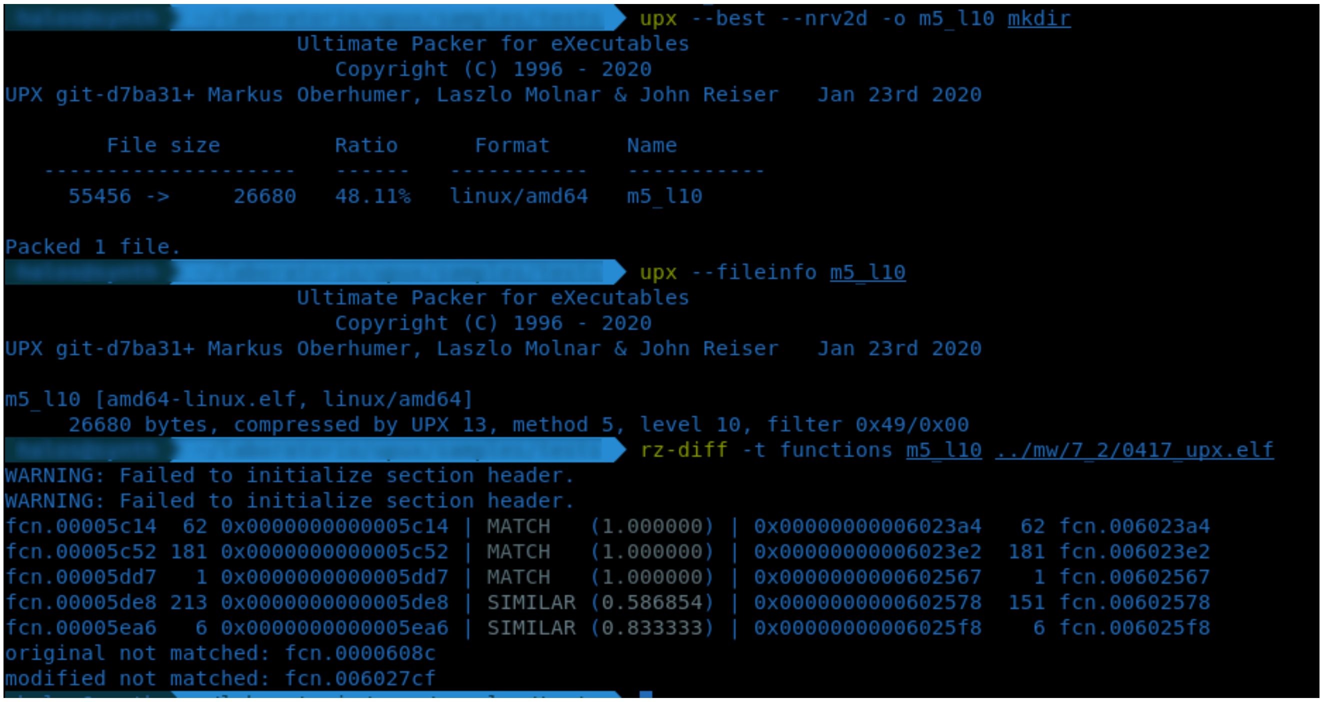
With the help of `upx --fileinfo` we got the parameters needed to pack another executable with almost the same compression and decompression algorithm.

```
> upx --fileinfo ../mw/7_2/0417_upx.elf
                    Ultimate Packer for eXecutables
                       Copyright (C) 1996 - 2020
UPX git-d7ba31+ Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

../mw/7_2/0417_upx.elf [amd64-linux.elf, linux/amd64]
    11756 bytes, compressed by UPX 13, method 5, level 10, filter 0x49/0x01
```

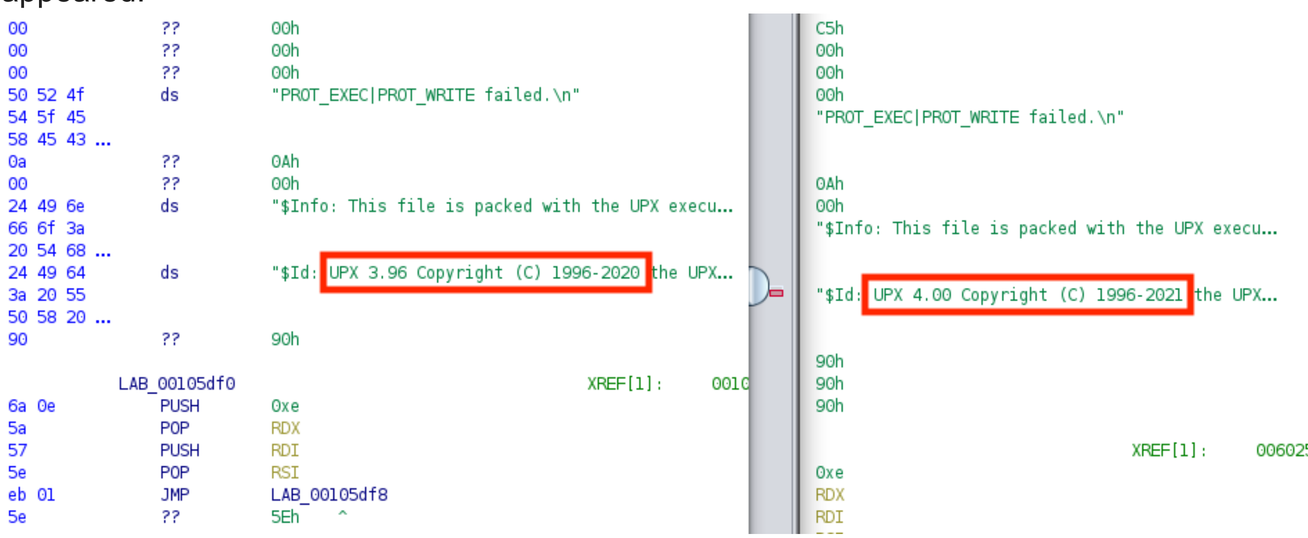
Extracting compression information from a malware sample.

To compress the sample, the attackers used a command similar to `upx --best --nrv2d <elf_file>` . As a starting point to check differences, we used the `rz-diff` tool to compare the main decompression functions:



Creating an executable with similar packing and comparing unpacking functions.

We started comparing the differences between the functions, looking for the code we were thinking the attackers added to the decompression process. An unexpected difference appeared:



Malware sample packed using UPX 4.0.0.

At the moment, the stable UPX version is v3.96, and version 4.0.0 is in development. Changelog doesn't seem to contain big changes in how ELF compression works, but there are a lot of commits that affect portions of the code involved in the calculation of these values.

We then checked how this new version handled the issues we were seeing by downloading the pre-release version of UPX and compiling it. After only fixing the `UPX!` strings headers (`b_info` and `p_info` structures were left untouched) and passing this executable to `UPX` version 4.0.0, the sample was accurately decompressed.

```
[redacted]/samples/mw/7_2 ../../../4/src/upx.out -t 0417_upx.elf
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2021
UPX git-a46b63  Markus Oberhumer, Laszlo Molnar & John Reiser    Jan 1st 2021

testing 0417_upx.elf [OK]

Tested 1 file.

WARNING: this is an unstable beta version - use for testing only! Really.
```

Successful extraction with UPX 4.0.0 (commit a46b63).

It is possible that the attackers realized that this version of UPX (which is still in development) generates functional samples that cannot be extracted by standard production versions of UPX versions used by default by everyone.

Universal Manual Unpacking

Instead of digging deeper into the modifications introduced by attackers, there is another approach we can consider. The idea here is to stop the debugging process when the code and data are already unpacked but the unpacked code hasn't been executed yet, to prevent the subsequent possible data and code modifications, and write down the unpacked code and data back to the disk. This approach is widely used to unpack Windows samples, but what about IoT threats? From a high-level perspective, the same logic can certainly be applied to them.

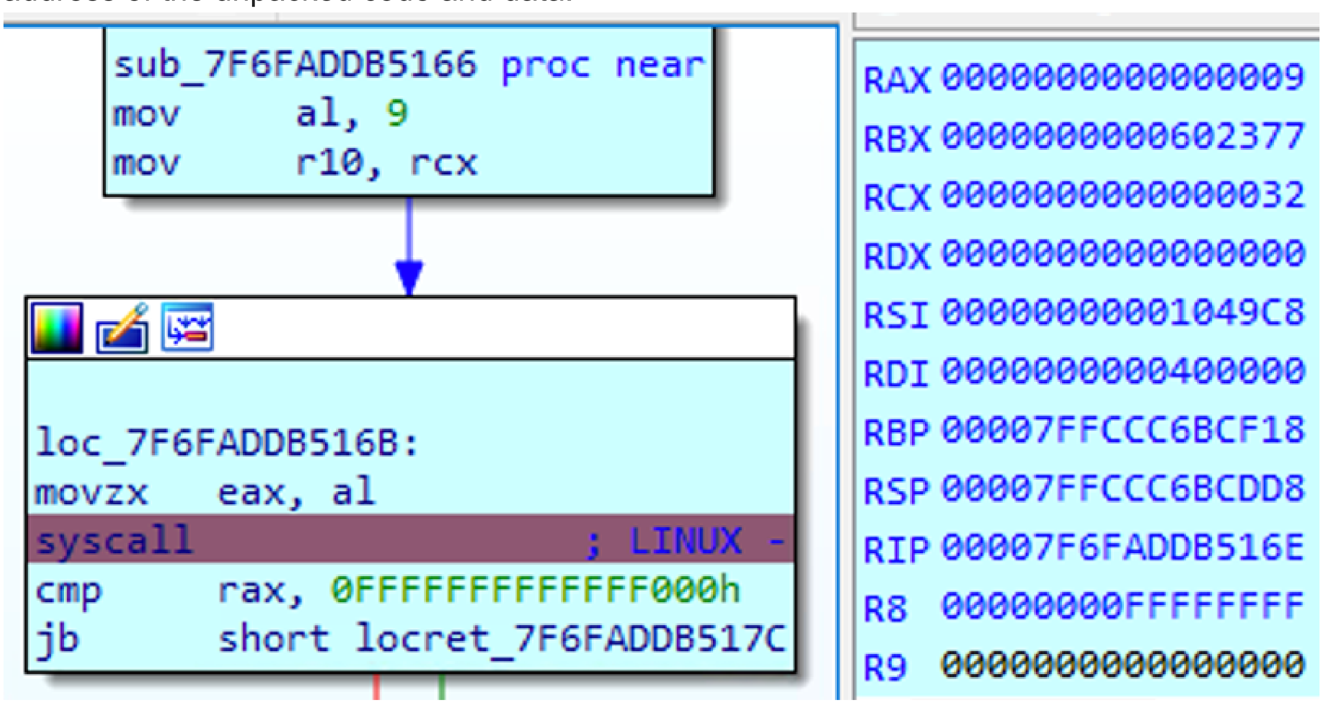
Here are several universal techniques that allow us to circumvent packers regardless of what modifications are introduced. They generally involve relying on the steps that an unpacker has to do in order to achieve its goal, mainly:

1. The packed code and data should be read and unpacked
2. A big memory block should be available to host the resulting unpacked sample
3. The result of the unpacking should be written into this big memory block
4. Depending on the existing protection flags for this block, they may need to be adjusted to make code execution possible
5. Finally, the control should be transferred to the first instruction of the unpacked code known as the Original Entry Point (OEP)

So, how can these techniques help us unpack the samples?

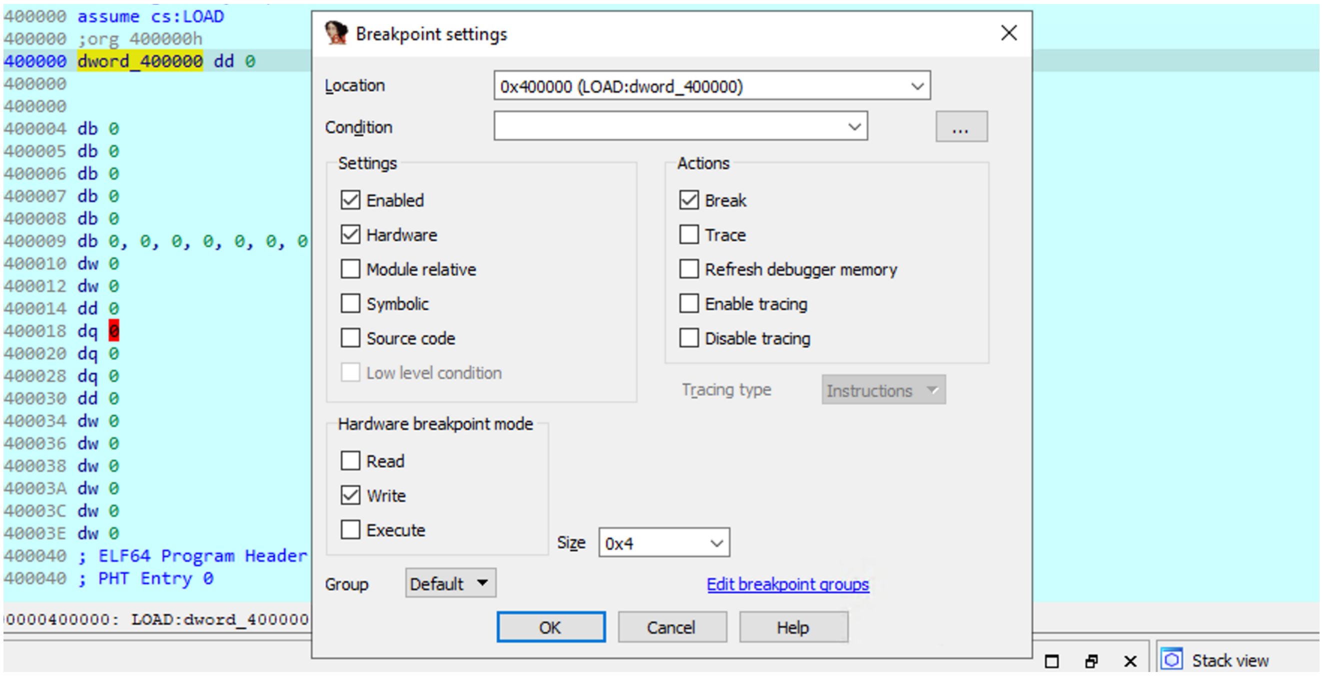
1. Generally, packed code and data have high entropy and can therefore be easily identified in hex editors. Finding these blocks and tracking their cross-references can help us find the decryption/decompression/decoding routine(s).

2. Keeping track of memory allocations (mmap syscall) may help us find the future virtual address of the unpacked code and data.



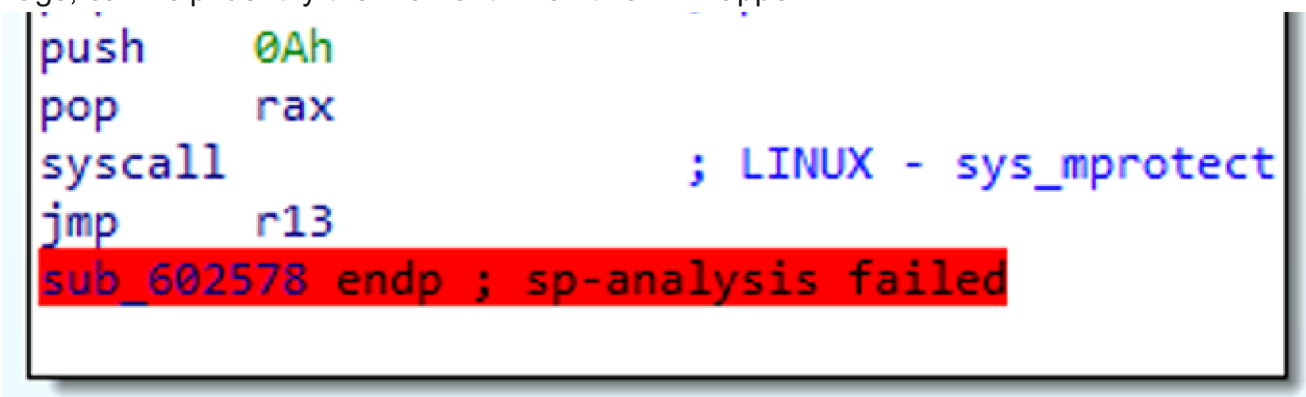
mmap syscall (rax = 0x09) with a big memory block length requested.

3. Memory or hardware breakpoint on write operation set on the allocated block will help us intercept the moment when the unpacked code and data of interest will be written there.



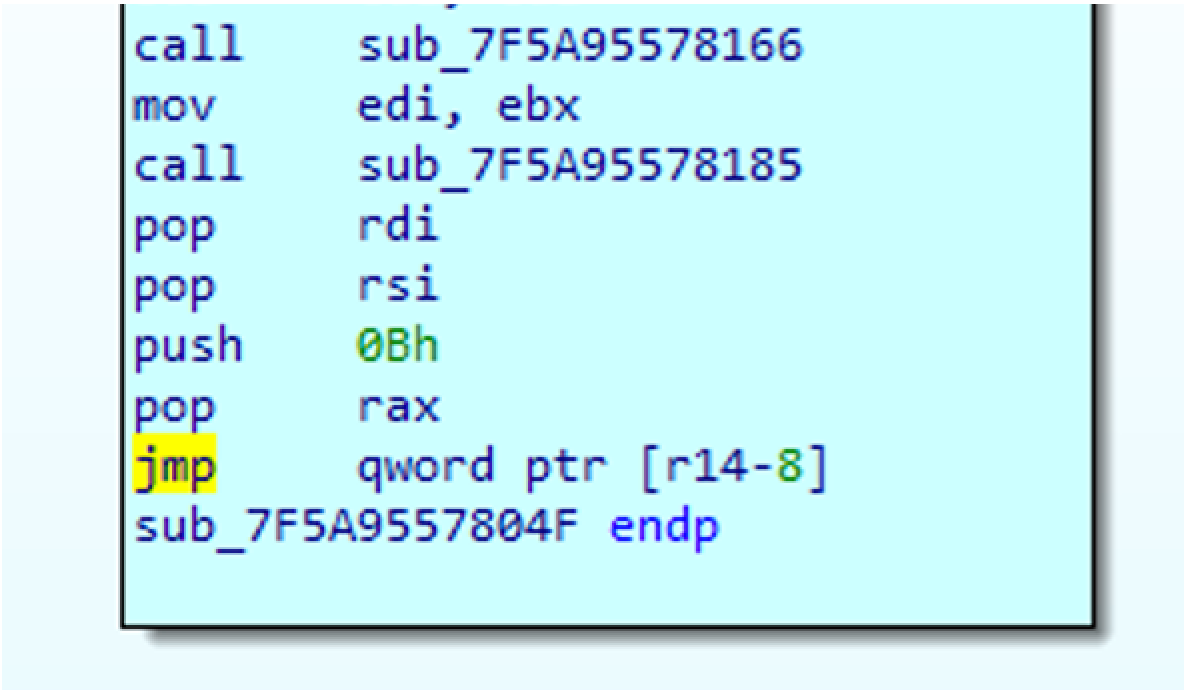
Setting a hardware breakpoint on write operation in IDA.

4. Keeping an eye on the mprotect syscall, which is commonly used to change protection flags, can help identify the moment when this will happen.



mprotect syscall followed by an unusual control flow instruction.

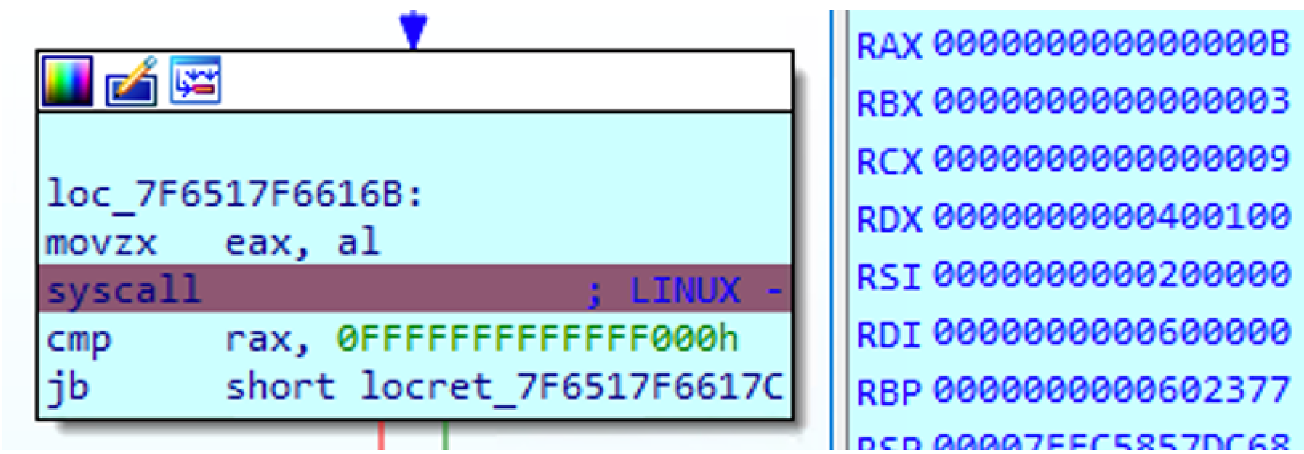
5. Looking for unusual control flow instructions can help identify the moment when the unpacker finished its job and is ready to transfer control to the first instruction of the freshly unpacked code (OEP).



The final unusual control flow instruction leading to the OEP.

Following these approaches separately or in combination eventually helps intercept the moment when the unpacked code and data finally reside in memory readily available to be dumped to the disk for subsequent static analysis.

In addition to these techniques, calling munmap syscall next to transferring control to the OEP is another feature of UPX that allows researchers to quickly unpack such samples. They can simply intercept it and then follow the execution flow.



munmap syscall (rax = 0x0B) executed next to transferring control to the OEP.

Conclusions

The landscape of malware targeting the IoT sector keeps evolving and borrowing many features from the more well-known IT sector. Staying up-to-date with the latest trends in this area and being able to handle them helps the security community combat new threats more efficiently and reduces the potential impact of associated cyberattacks.

Related Content

RESEARCH REPORT

OT/IoT Security Report – 2021 2H

- Insights on the latest cyber threats
- How and why critical infrastructure industries such as transportation and healthcare are being targeted
- Analysis of recent ICS-CERT vulnerabilities and exploitation trends
- Remediation strategies to address today’s emerging threats

Download

Related Links

- Blog: Extract Firmware from OT Devices for Vulnerability Research
- Blog: Critical Log4shell (Apache Log4j) Zero-Day Attack Analysis
- Blog: How to Analyze Malware for Technical Writing
- Blog: The Clever Use of Postdissectors to Analyze Layer 2 Protocols
- Blog: Nozomi Networks Labs Webinars & Podcasts

Nozomi Networks Labs

Nozomi Networks Labs is dedicated to reducing cyber risk for the world's industrial and critical infrastructure organizations. Through our cybersecurity research and collaboration with industry and institutions, we're helping defend the operational systems that support everyday life.