FEATURE 1

EXPLORING THE EVOLUTIONARY PATTERNS OF TIBS-PACKED EXECUTABLES

Rachit Mathur, Aditya Kapoor
McAfee, USA

This year we have seen a very large number of packed executables related to W32/Nuwar, aka the Storm worm, all of which have used a packer commonly known as Tibs. Broadly speaking, Tibs is a polymorphic closed source packer that is used by its author(s) to obfuscate a variety of malware. All the malware we have seen packed with Tibs to date has been motivated by monetary gains, primarily involving spam.

Tibs packed executables evolve continually, thus allowing the malware to pass undetected through some anti-virus defences. This article presents an analysis of the techniques used by the Tibs packer and describes the reasons for its prolonged effectiveness. (Note: the terms 'packing' and 'encryption' are used interchangeably in this article.)

1. PROLIFERATION TACTICS

The 'Tibs gang' has been very successful in its use of social engineering – luring and tricking large numbers of users into downloading and executing its malware. We have seen downloaders, worms, mass mailers, proxy agents and spam-mailbots all packed with Tibs.

The Tibs gang uses a range of tactics to attempt to penetrate security defences at multiple levels:

- In an attempt to evade spam filters, the text of the emails in which malware is sent is modified frequently.
- To avoid network traffic recognition, variations are introduced in encrypted Overnet traffic.
- To defeat analysis tools used by cautious administrators, the malware installs kernel mode rootkits to hide files, processes etc. In order to minimize their footprint in the registry some variants infect binaries that are loaded at startup. The variant inserts its own loader code into the victim binary thus ensuring the malware will be loaded on system startup.
- To avoid detection by AV scanners the server hosting the malicious binary produces modified executables every so often (approximately every 15 minutes).

Figure 1: Google map – infected nodes.

In order to harvest samples from the servers hosting Tibs-packed files, we monitored thousands of IP addresses for a period of time. The list of IP addresses was updated continually with new links being added while dead links were dropped. Figure 1 is a snapshot of the Geo-Mapping of the IP addresses hosting these Tibs-packed files. (This information is not completely representative of the threat; however, it provides an approximate idea of where those executables were hosted at a point of time.)

Many of the aspects of this threat have already been discussed in *Virus Bulletin* [1, 2] and elsewhere [3–5]. This article adds to the previous articles by discussing the workings of the Tibs packer.

2. TIBS PACKER OVERVIEW

Tibs executables are packed polymorphically, i.e. the decryptor code differs among variants. However, the polymorphic engine is not contained within these executables, which means they do not have the ability to generate polymorphic variants on their own. This is where Tibs differs from the traditional notion of polymorphic malware, and its behaviour falls instead under what is commonly known as 'server-based polymorphism' – the server hosting the malware returns executables with polymorphic variations in the decryptor code when queried at different points in time.

Figure 2 is a diagrammatic representation of a typical Tibs-packed executable containing a 'server-side' polymorphic decryptor and the encrypted malcode. The underlying code may be a pure form of malware or encrypted either by a flavour of TEA [6], UPX etc. or by a combination of these.

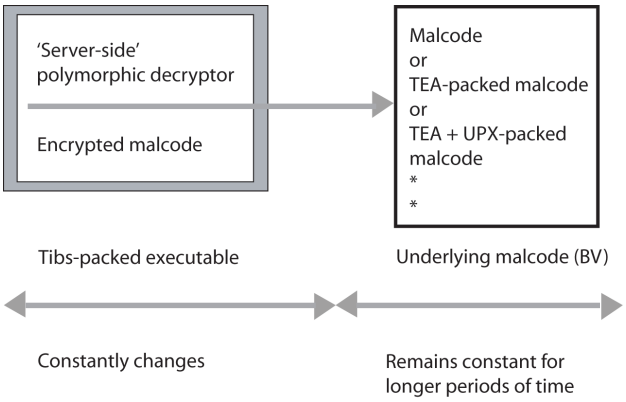

Figure 2: Typical Tibs-packed malcode.

Tibs-packed samples implement simple yet effective code transformations in their decryptors to hinder detection. Normally the decryptor code is fairly small and the code bytes of the decryptor are modified frequently, while the decryptor logic and underlying decrypted code (base variant) is changed less frequently – in regular polymorphic behaviour the decrypted code simply remains constant. It is the server-side nature of Tibs that allows the VXers to manipulate the underlying code as well.

The decryption steps of Tibs are outlined in Figure 3.

1. Locate the start address of encrypted data and size/end of the data
2. Calculate key(s): key[i]
3. Apply key(s)
4. Transfer control to decrypted code

Figure 3: Decryption steps.

The obvious first step is to locate the beginning and end of the data that needs to be decrypted. Then the key(s) need to be identified – typically there are two. Thereafter the key is applied to the encrypted data, one dword at a time, and finally control is transferred to the decrypted code. Although the decryption steps of many decryptors are the same as those shown here, the evolutionary trends of the decryptor code and the decryption algorithm itself are interesting in the case of Tibs.

3. TIBS EVOLUTION PATTERNS

The evolutionary trends of the Tibs polymorphic decryptor can be identified by analysing the differences between the executables as they change on the server hosting them. The morphing techniques used in the decryptor can be classified according to the frequency with which they are applied (high, medium or low frequency).

3.1 High-frequency morphing techniques

Here, at least one of the keys changes frequently and the executable is recompiled. Since the key is changed, the bytes of the entire encrypted data change, and this makes up the majority of the body of the executable. The decryption algorithm and the decryptor code remain the same except for the key.

This change is introduced a couple of times every hour to produce a new file from the server hosting the malicious executables.

3.2 Medium-frequency morphing techniques

These changes are introduced once every couple of days and involve the application of various code-morphing and anti-emulation techniques. The decryption algorithm remains the same but the code changes.

Some of the transformations that may be introduced are as follows:

a. Use of MMX instructions: code morphing using MMX instructions can be applied as shown in Figure 4.

```
                          movd   mm7,edx
mov [esi], edx →          movq   mm3,mm7
                          movd   [esi],mm3
```

Figure 4: MMX transform.

b. Use of fake *Windows* API (WAPI) calls: fake calls may be introduced to *Windows* functions such as 'CreateMDIWindowA', 'ILGetSize', etc. These API calls are fake because they are not called to perform the actual purpose for which they exist. Instead, null or junk parameters are passed and the returned values are validated during decryption. These return values (which are mostly *Windows* standard error codes) are typically used as one of the keys during decryption. For example, the SHFindFiles function displays the search window user interface if called 'properly', but the malware makes this call with null parameters and without calling CoInitialize, resulting in the error code 0x800401f0. This is then used as one of the decryption keys.

c. Other techniques such as register renaming, CFG obfuscation, dead code insertion, replacing SESE

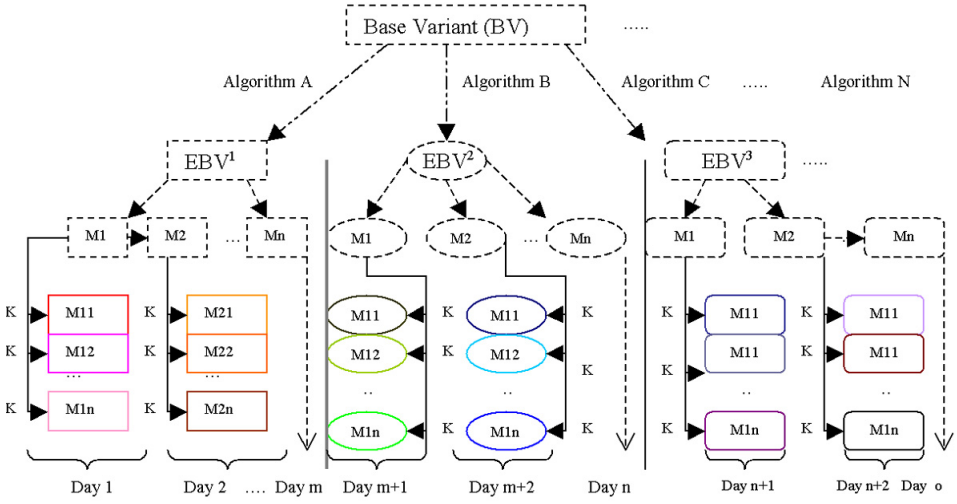

Figure 5: Evolution of a base variant.

(Single Entry Single Exit) blocks with semantically equivalent code, converting simple calculations into time-consuming loops etc. may also be introduced.

3.3 Low-frequency morphing techniques

Here, the length of time between changes can be anything from a week to over a month. In low-frequency transformations it is the apply-key step (step 3 in Figure 3) that changes – i.e. the decryption algorithm changes semantically. The decryption algorithm is generally fairly simple. Some examples of algorithms applied to encrypted data one dword at a time, are:

- dword + K
- (dword + K1) ^ K2
- rotate ((dword + K1), K2)
- a = RTC(dword, K1) -> 'modify carry flag' -> (RTC(a, K1) + K2) ^ K3 , RTC = rotate through carry
- (dword / K1) ^ K2

Once the obfuscations mentioned in section 3.2 are applied, it switches back to just changing the key in the resulting code for the next couple of days and applies the medium frequency transformation again. This cycle can continue for anything from a week to several months and then the low frequency transformation is applied.

Figure 5 is a pictorial representation of the evolutionary trend of Tibs executables. A base variant is encrypted using an algorithm (Section 3.3) to give an encrypted base variant (EBV). The different shape of these variants represents the semantic non equivalence of Tibs decryptors. Thereafter, the transformations from Section 3.2 are applied to obtain mutants (M*). K represents a random key that is chosen for the mutant that gets released. The dotted lines represent virtual intermediary steps, while solid lines represent the mutants that are released. Different colours represent different mutants. The time line increases from left to right and the granularity of the high-frequency key change is approximated as one day.

4. TIBS DETECTION TREND

Figure 6 presents detection statistics for a randomly chosen set of 60 Tibs-packed samples obtained during a period of approximately one month. 26 static signature-based scanners were tested against the samples. Note that the scan result for each sample was obtained as soon as the sample was downloaded from the malicious host, with the latest scanner signatures available at that time.

Figure 6 presents the total, as well as accurate (signature-based) detection counts, where the total also includes heuristic detections.

On some of the days the detection rates were better than on others because there had been no significant change in the malware. In fact, there are hundreds of samples with different hashes that appear daily, yet Figure 6 represents a satisfactory test as there is no significant

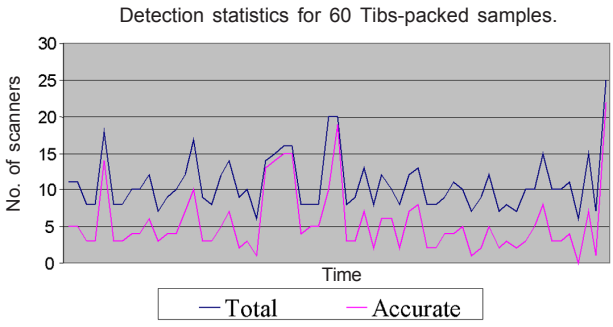

Figure 6: Total and accurate (signature-based) detection counts.

change in the variants that appear within a day (Section 3). The detection rate in a day for most AV vendors is fairly static. It is evident from Figure 6 that the number of accurate detections is low when compared to heuristic detections. Furthermore, no AV vendor showed consistently accurate detection for all samples.

5. DETECTION CHALLENGES

For the hundreds of different samples generated every day as described in Section 3, a byte-based signature could be written that detects on the decryptor loop code itself (which remains the same except for the key value). However, this would not be effective for longer than a day or two because the code-morphing techniques described in section 3.2 would be introduced to obfuscate the code and muddle the byte patterns.

To handle this mutation one could use emulation, which is a popular way of dealing with polymorphism. The loop could be emulated to decrypt the underlying data, and detection could be achieved based on that decrypted data. This may provide detection for a longer period of time, depending on how robust the emulator is. However, as mentioned in section 3.2, Tibs introduces anti-emulation techniques along with obfuscation. For example, some emulators may not be able to handle MMX instructions. While emulators handle most common WAPI calls to facilitate sufficient emulation of code, handling all WAPI calls – i.e. figuring out the number of parameters that each takes and the return value(s) depending on the context of the call – becomes increasingly challenging.

In order to achieve accurate detection of Tibs-packed threats for longer periods of time, one could choose to base detection on attributes that change less frequently, such as the underlying code. Understanding the decryptor logic and using better methods to decrypt could be one way to add generic detection. This could be achieved by using cryptanalysis on the encrypted code. Alternatively, a detection technique may choose not to decrypt and leverage the fact that the encryption is always one dword at a time by performing statistical analysis on the encrypted data. However, one of the major concerns for AV developers with such techniques is efficiency; the desktop scanner's speed should be acceptable to end-users and such cryptanalysis techniques tend to slow performance significantly.

Heuristics based on file geometry can also be used to detect on the overall structural commonalities of these executables. Attributes such as file size, number/names of sections, section flags, linker versions and unusual imports may serve as good aids in writing detections for these samples heuristically. The risk with such approaches, of course, is that false alerts may be produced on clean files.

The server-side aspect of this polymorphic approach creates the opportunity for blending automated sample generation with periodic human intervention, thus making such threats more insidious than their traditional counterparts.

With Tibs being a proprietary packer, it is tricky to guess how much of its polymorphic process is automated. In theory, a lot of it could be automated, but we do not know how much of it is in reality – this could be an interesting piece of research. The minimum requirement for any detection signature is that it should detect all samples that are generated automatically.

CONCLUSION

This article describes a trend in the evolutionary pattern of Tibs-packed malware and discusses various detection techniques and their pitfalls. The approach described in this article is not the only way in which the server-based malware model can work and this threat may change its tactics in future. There is room for improvement in both the attack and defence techniques and the bar will be raised on each side as this battle progresses.

The authors of Tibs are not the first to use server-based obfuscation techniques, but they are surely amongst the most successful with it. Other threats are likely to follow in its footsteps; we can expect a significant rise in the number of malware samples as the popularity of such techniques will almost certainly increase in the future.

REFERENCES

[1] Florio, E.; Ciubotariu, M. Peerbot: catch me if you can. Virus Bulletin, March 2007, p.6.

[2] Bureau, P.-M.; Lee, A. Malware storms: a global climate change. Virus Bulletin, November 2007, p.12.

[3] McAfee VIL. W32/Nuwar@MM, http://vil.nai.com/vil/content/v_140835.htm.

[4] Wikipedia. Storm Worm, http://en.wikipedia.org/wiki/Storm_Worm.

[5] Boldewin, F. Peacomm.C Cracking the nutshell. http://www.reconstructer.org/.

[6] Wikipedia. Tiny Encryption Algorithm http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm.