TitanMist: Your First Step to Reversing Nirvana| TitanMist

mist.reversinglabs.com

Contents

Introduction to TitanEngine

One of the greatest challenges of modern reverse engineering is taking apart and analyzing software protections. During the last decade a vast number of such shell modifiers have appeared. Software Protection as an industry has come a long way from simple encryption that protects executable and data parts to current highly sophisticated protections that are packed with tricks aiming at slow down in the reversing process. Number of such techniques increases every year. Hence we need to ask ourselves, can we keep up with the tools that we have?

Protections have evolved over the last few years, but so have the reverser's tools. Some of those tools are still in use today since they were written to solve a specific problem, or at least a part of it. Yet when it comes to writing unpackers this process hasn't evolved much. We are limited to writing our own code for every scenario in the field.

We have designed *TitanEngine* in such fashion that writing unpackers would mimic analyst's manual unpacking process. Basic set of libraries, which will later become the framework, had the functionality of the four most common tools used in the unpacking process: debugger, dumper, importer and realigner. With the guided execution and a set of callbacks these separate modules complement themselves in a manner compatible with the way any reverse engineer would use his tools of choice to unpack the file. This creates an execution timeline which parries the protection execution and gathers information from it while guided to the point from where the protection passes control to the original software code. When that point is reached file gets dumped to disk and fixed so it resembles the original to as great of a degree as possible. In this fashion problems of making static unpackers have been solved. Yet static unpacking is still important due to the fact that it will always be the most secure, and in some cases, fastest available method. That is why we will discuss both static and dynamic unpackers. We will also see into methods of making generic code to support large number of formats without knowing the format specifics.

TitanEngine can be described as Swiss army knife for reversers. With its 400 functions, every reverser tool created to this date has been covered through its fabric. Best yet, *TitanEngine* can be automated. It is suitable for more than just file unpacking. *TitanEngine* can be used to make new tools that work with PE files. Support for both x86 and x64 systems make this framework the only framework supporting work with PE32+ files. As such, it can be used to create all known types of unpackers. Engine is open source making it open to modifications that will only ease its integration into existing solutions and would enable creation of new ones suiting different project needs.

TitanEngine SDK contains:

- Integrated x86/x64 debugger
- Integrated x86/x64 disassembler
- Integrated memory dumper
- Integrated import tracer & fixer
- Integrated relocation fixer
- Integrated file realigner
- Functions to work with TLS, Resources, Exports,…

Introduction to TitanMist

Security is notoriously disunited. Every year multiple tools and projects are released and never maintained. TitanMist is its inverse opposite. Built on top of TitanEngine, it provides automation and manages all known and good PEID signatures, unpacking scripts and other tools in one unified tool. TitanMist is the nicely packaged and open source catch all tool that will become your first line of defense. The project also goes beyond pure tool development. It builds a forum to share information and reverse engineering experience built around the biggest online and collaborative knowledge base about software packers.

With the increase in packed and protected malicious payloads, collaboration and quick response between researchers has become critical. As new sample numbers are quickly closing to 40M samples per year, solution to this problem has to come from reverse engineers themselves, integrating the work that they have done in the past and they continue to do. Huge databases of format identification data and unpacking scripts can be reused in a way to maximize automation. Yet, where do we find a definite collection of functional tools, identification signatures and unpacking tools? And how do we integrate them in a meaningful and accurate way?

TitanMist approaches these problems in a manner recognizable to every reverse engineer. It aims to mimic, but automate, the reversing process enabling everyone to easily create unpackers and integrate them in an extensible system. This builds a powerful and fast growing community analysis tool. Overcoming the most basic problems of reverse engineering problems was the top priority for the TitanMist project. Hoping to bridge the programming knowledge barrier which troubles many reverse engineers TitanMist introduces a variety of programming languages in which unpackers can be written in.

TitanMist goes beyond languages that compile to native code relying heavily on popular and easy to learn script languages. Backed up by LUA and Python this project makes coding unpackers a much simpler task. However the challenge of making TitanMist as easy to adopt and extend as possible meant that the project has to go further than extending support for more programming languages. Knowing that most of reverse engineers are familiar with debugger level script language OllyScript we added the support for it as well. Combined with the full TitanEngine functionality these scripts become powerful automated unpackers which combined with the layer of file format identification create a unique database of file analysis tools.

There is hardly a software protection nowadays that has only a single layer of code containing the whole stub code. Even some software packers such as PeCompact implement multiple layers in the process of software decompression. It is common for these additional layers to do the most interesting protection operations, such as memory decompression, import table processing and entry point protection and redirection. Therefore in order to fully dynamically unpack these kinds of protections we need to move through the layers and collect the needed data along the way. The protection we have chosen to analyze is AlexProtector because it uses a multiple layer protection model along with other interesting protection features. During analysis of this protection, we will encounter: obfuscations, antidebugging, antitracing, antiemulation, checksum checks, import elimination and stolen bytes at the entry point. Quite an impressive list of protections found in software protection from 2004, which is why we at the ReversingLabs commonly use it as an introduction to more complex protection solutions.

The entry point of the packed file gives us a vague clue on how interesting this analysis will be. It looks like this:

```
  PUSHAD
  CALL L002
L002:
  POP EBP
  SUB EBP,00401006
  CALL 00407036
  JMP L007
  DB E9
L007:
  MOV EAX,DWORD PTR SS:[ESP+C]
  JMP SHORT 0040701E
```

The usual method of getting the code offset delta via CALL/POP, followed by simple obfuscations and the first packer controlled exception. Luckily for us the whole first protection layer is unprotected, so basic stub functions are accessible from start. They include antidebugging functions and more importantly, the decompression of the main protection layer and the transfer of control to it. This first layer's primary point of interest is the layer control transfer. To locate that part of the code we scroll down a bit and spot the aplib decompression code. That recognizable code:

```
  PUSHAD
  MOV ESI,DWORD PTR SS:[ESP+24]
  MOV EDI,DWORD PTR SS:[ESP+28]
  CLD
  MOV DL,80
  XOR EBX,EBX
  MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
...
  SUB EDI,DWORD PTR SS:[ESP+28]
  MOV DWORD PTR SS:[ESP+1C],EDI
  POPAD
  RET
```

It is located at offset +0x108 relative to the entry point. Since it is common for protectors to compress and/or encrypt their layers, setting a hardware breakpoint here seems like a good place to start. After a few exceptions we finally hit it, and we can collect the needed data from this function's input parameters. If we take a look at the stack we can see these numbers:

```
0012FF50   00407AB9  RETURN to 00407AB9 from 00407108  ;Called from
0012FF54   00408531  00408531                          ;Packed layer content
0012FF58   00330000                                    ;Allocated layer buffer
```

If we look at the address from which the layer decompression was called, we see:

```
 CALL 00407108                                         ;aPLib decompression
 ADD ESP,8
 PUSHAD                                                ;AntiTracing via RDTSC
...
 ADD ESP,4
 RDTSC
 MOV EBX,EAX
...
 ADD ESP,4
 MOV ECX,EDX
...
 ADD ESP,4
 RDTSC
 SUB EAX,EBX
...
 ADD ESP,4
 SBB EDX,ECX
 RDTSC
 ADD EAX,EBX
...
 ADD ESP,4
 ADC EDX,ECX
 RDTSC
 SUB EAX,EBX
...
 ADD ESP,4
...
 ADD ESP,4
 SBB EDX,ECX
...
 ADD ESP,4
 TEST EDX,EDX
 JNZ SHORT 00407B36                                    ;Executed if code is traced
 POPAD
```

This antitrace code is repeated throughout the code many times, and it is also present in all API redirections. Following this is a single exception that is executed just before control is transferred to

second protection layer. Instruction JMP EDI, which is located just above the custom memory checksum algorithm, transfers the control to the next layer. That next layer starts like this:

```
  JMP L003
L001:
  JMP L004
  ???
L003:
  JMP L001
L004:
  CALL L036
  TEST AL,83
  LES EAX,FWORD PTR DS:[EAX+EDX*4]
  NOP
```

This protection layer holds all information necessary for unpacker coding in this layer. Imports are handled at these locations:

```
+0x9A5:  PUSH EBX
  LEA EDX,DWORD PTR SS:[EBP+401108]
  CALL NEAR EDX                                  ;decompress IAT data
  ADD ESP,8
  MOV EDI,DWORD PTR SS:[EBP+4023A9]
+0x9B7:  CMP BYTE PTR DS:[EDI],0C3
  JNZ 00330A8C
  ADD EDI,2
  PUSH EDI
  CALL NEAR DWORD PTR SS:[EBP+4024C8]            ;GetModuleHandleA
  TEST EAX,EAX
...
+0xA95:  INC EDI
  PUSH EDI
  PUSH DWORD PTR SS:[EBP+40247C]
  CALL NEAR DWORD PTR SS:[EBP+4024C4]            ;GetProcAddress
  XOR EBX,EBX
  PUSHAD
```

And this is where we have a choice to either place breakpoints at these locations or analyze a memory buffer holding IAT data. That buffer is decompressed just before it is processed. If we examine it, we see the following pattern:

```
typedef struct ALEX_IAT_DLL{
    BYTE DLLSignature; //0xC3
    BYTE DLLNameLength;
    // DLLName[DLLNameLength] followed by 0x00
}ALEX_IAT_DLL, *PALEX_IAT_DLL;
```

```
typedef struct ALEX_IAT_APIENTRY{
    BYTE APINameLength;
    // APIName[APINameLength] followed by 0x00
    BYTE RedirectionNumber; //Number of redirections
    DWORD RedirectionAddress[RedirectionNumber];
}ALEX_IAT_APIENTRY, *PALEX_IAT_APIENTRY;
```

Either choice is a good one. However AlexProtector uses a protection technique called "import elimination" which means that parts of the code section which are used as call gates to Windows API are damaged and must be fixed. This data is stored in the import structure we just analyzed. However there is a problem with this since we don't know the original location of the import table and we must reserve some space for it. The solution to this problem is to reserve space in the new section but a detailed description on how to do this will be done next week in part two, when we create the unpacker for AlexProtector.

Solving the problem of imports leaves us with just one more problem, the entry point. After some quick tracing we find this code part:

```
+0x107B:  MOV ESI,DWORD PTR SS:[EBP+402385]          ;SS:[00408385]=004012E6
  ADD EAX,EBX
  MOV BYTE PTR DS:[EAX],0E9                          ;Write jump to EP
  INC EAX
  MOV ECX,ESI
...
+0x11E5:  MOV DWORD PTR SS:[ESP+1C],EAX
  POPAD
  JMP NEAR EAX                                       ;Jump to stolen EP code
...
  MOV EDI,DFB4AF72
  LEA EDI,DWORD PTR DS:[1A58BA5F]
  DEC EDI
  SHRD EDI,ESI,0F2
  BSR EDI,ESI
  TEST EDI,2730DC5C
  TEST EDI,874CDC1C
  AND EDI,9378C643
  JMP 004012E6                                       ;True jump to EP
```

The first part of this code at +0x107B writes the jump to entry point, more specifically the first instruction after a few instructions that have been stolen. Since those stolen instructions must be executed, the protector creates yet another layer which is executed just before the jump to entry point. That layer contains the stolen instructions, and a jump at the end of that code leads to the protected file code section. Stolen instructions are mixed with the junk instructions so it's easiest to dump this layer to new sections and fix the jump to entry point. This was the last piece of information needed to create an unpacker for this protection. Choices of various script languages or native code programming languages for creating one are before us. You can find the source code for these within the TitanMist package.

References and tools

[1] TitanMist – **http://mist.reversinglabs.com/**
[2] Knowledge base – **http://kbase.reversinglabs.com/**