Fast & Furious reverse engineering | TitanEngine

titan.reversinglabs.com

REVERSING LABS | Reverse Engineering & Software Protection

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 reversers 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 250 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 static unpackers

Most of basic unpackers are of the static variety. We take this observation very loosely as this depends on the complexity of the format being unpacked. In most cases writing such unpackers is easy because the format being unpacked is a simple one or more commonly referred to as a crypter.

This kind of PE file protectors (because packing is a very basic form of protection) have a simple layout that only encrypts the code and resources, and in some cases even takes the role of the import loader. Even if we encounter the most advanced representative of this shell protection type it won't differ much from its most basic protection model. Which is, no modification to the PE section layout other than adding a new section for the crypter code and encryption of the entire code and resource sections with possible import loader role for the crypter stub. Since these modifications don't impact the file in such way that major file reconstruction should be done writing static unpackers also has its general model. This is, get the needed data for decryption of the encrypter parts and reconstruction of the import table followed by removing the crypter section.

With the slight variations of the guidelines described above this could be considered as the basic crypter model. These variations could be variations in the position of the crypter code, way it handles imports and some protection shell specifics such as: protected entry point, import redirections or eliminations, code splices, code markers, etc.

However static unpackers can be used for a more difficult use cases which require the full file reconstruction in order to complete the unpacking process. In such cases static unpacking can be used and it's recommended only if the security is of the vital importance. These cases most commonly require the identification of the compression algorithm used and its adaptation to our own code. This code ripping must be done very carefully and it requires the full understanding of the algorithm which decompresses the code. There are a few standard compression algorithms in use by most PE shells so we can use this to create our own database of corresponding decompression algorithms to ease the unpacker writing process. No matter which path we choose we must always check whether or not the algorithm has changed since this is one way to tamper with the unpackers. Dynamic unpackers are resilient to such changes.

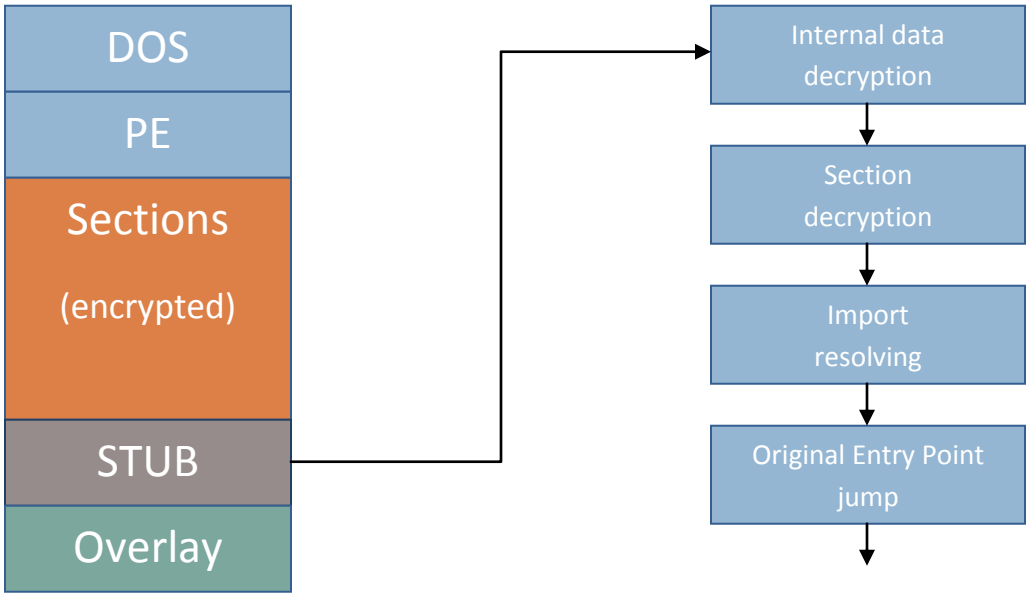


Figure (1) *Crypter file & execution layout*

Introduction to dynamic unpackers

Most common unpacker type is dynamic. This model is widely used because it is easy to implement and resilient to minor changes in packing shell decryption and decompression algorithms. But due to the fact that files do execute during unpacking process this method must be conducted with great care about system security. There is a risk of files being executed outside the unpacking process or even stuck in infinite loops. This can and must be avoided by the unpacker implementation. Most logical choice is creation of internal sandbox for the unpacking process itself.

Dynamic unpacking is used on specific shell modifier types. These types have a more complex layout and file content is commonly not only encrypted but compressed too. To avoid heavy coding to allow what is basically recompiling of the file we execute it to the original entry point which is the first instruction of the code before the file was protected. Even though all shells can be dynamically unpacked this kind of unpacking is only used on packers, protectors, bundlers and hybrids.

Basic layout of such shell modifiers includes compression of the file sections and vital data and optionally their protection by encryption. Stub is usually located in the last section and section layout before packing is either preserved or all sections are merged into one. First case usually implies that each section still contains its original data only compressed while the second one implies a physically empty section which only serves as virtual memory space reserve. In this second case compressed data is usually stored inside stub section and upon its decompression returned to original location.

When creating dynamic unpackers it is important to always keep control over executing sample. At no point this control must be left in a gray area in which we are uncertain what will occur. Furthermore unpacking must be conducted inside safe environment so software sandbox must be designed. This along with multiple checks before and during the unpacking process ensures that we retain maximum control during this risky process. This kind of unpackers has a standard model which will be described in *Dynamic unpacker layout*. That kind of unpacker is a basic unpacker type that can be created with *TitanEngine* whose largest number of functions is dedicated to writing.

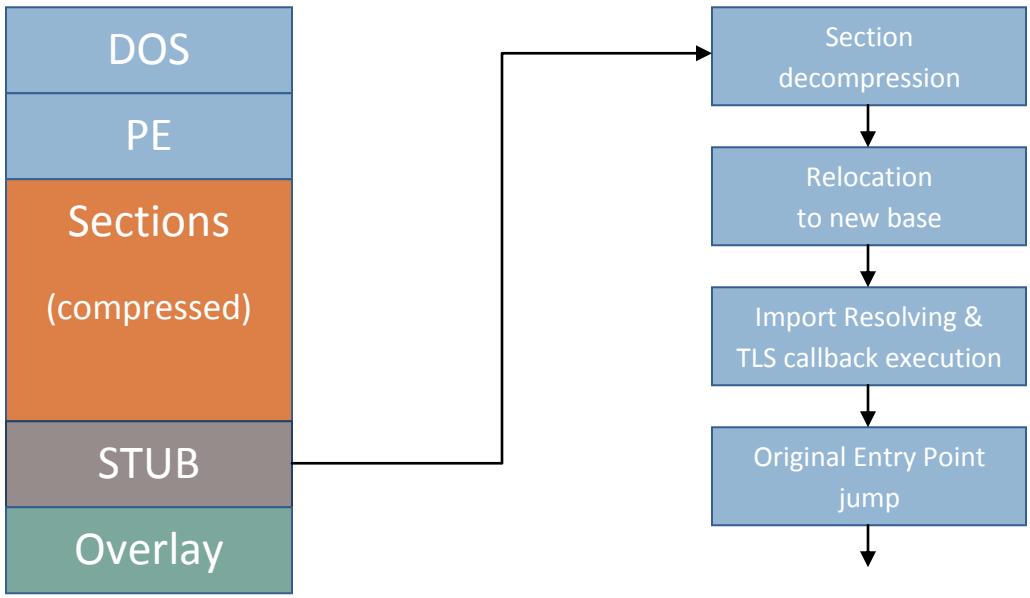


Figure (2) *Packer file & execution layout*

Most complex way of creating unpackers is creating generic unpackers. Totally opposite from the other two cases when creating generic unpackers you don't need to worry about extracting a good enough patter on code segment to create a good signature for your unpacker. Quite simply because these unpackers don't care about the shell specifics, they only care about their overall behavior which is common for shell modifiers of the same group. This means that there can never be a general generic unpacker but several wide range generic unpackers targeting specific behavior groups.

Here we will focus only on generic unpacking of packed executables and present a wide generic algorithm targeting these shell modifiers. Major challenge here is retaining as much control as possible without slowing down the unpacking process drastically. Slowdown occurs because we use memory breakpoints to monitor packed shell access to executable sections. If we reset the memory breakpoint each time the packer accesses the section we will have a major speed impact and if we don't we reset we risk not to catch the original entry point jump event and even let file execute. There are a few ways to do this but one is most common.

Generic unpackers commonly use WaitForDebugEvent timeouts to reset the breakpoints once one such breakpoint has been hit. Memory breakpoints can be hit for three reasons: when memory is read from, written to or executing. Since we only place memory breakpoints on places where we expect entry point to be we are only interested in execution case. To check whether or not that memory is executing we just check the EIP register to see if it matches our region. If it does that memory is executing. However we can set a breakpoint on the section which contains packer code. That is why we will track if the section has been written to prior its execution. If it has been written to it is highly possible that the entry point resides there and that we are at the right spot. We can track writing with memory breakpoints but one can also check that section hash just to make sure that the data has actually changed. Last check that should be performed before determining that we are sitting on a possible entry point is a check for known simple redirections that packers such as ASPack use to fool generic algorithms. Once we verify this last thing we can be fairly certain that we have found our entry point. To finish the unpacking we must dump and process imports which can be done with *TitanEngine*.

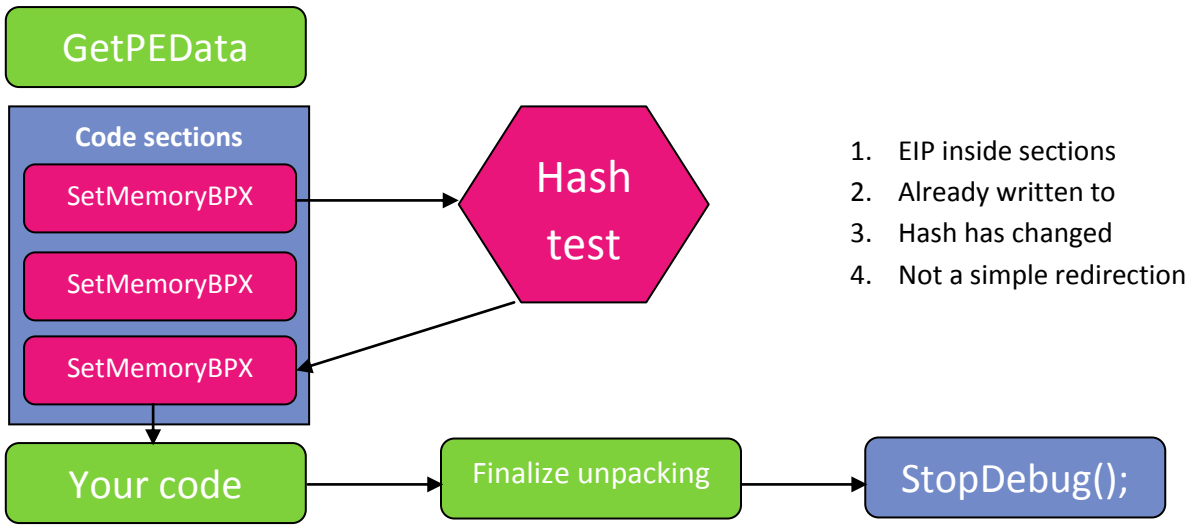


Figure (3) *Generic unpacker algorithm layout*

Dynamic unpacker layout

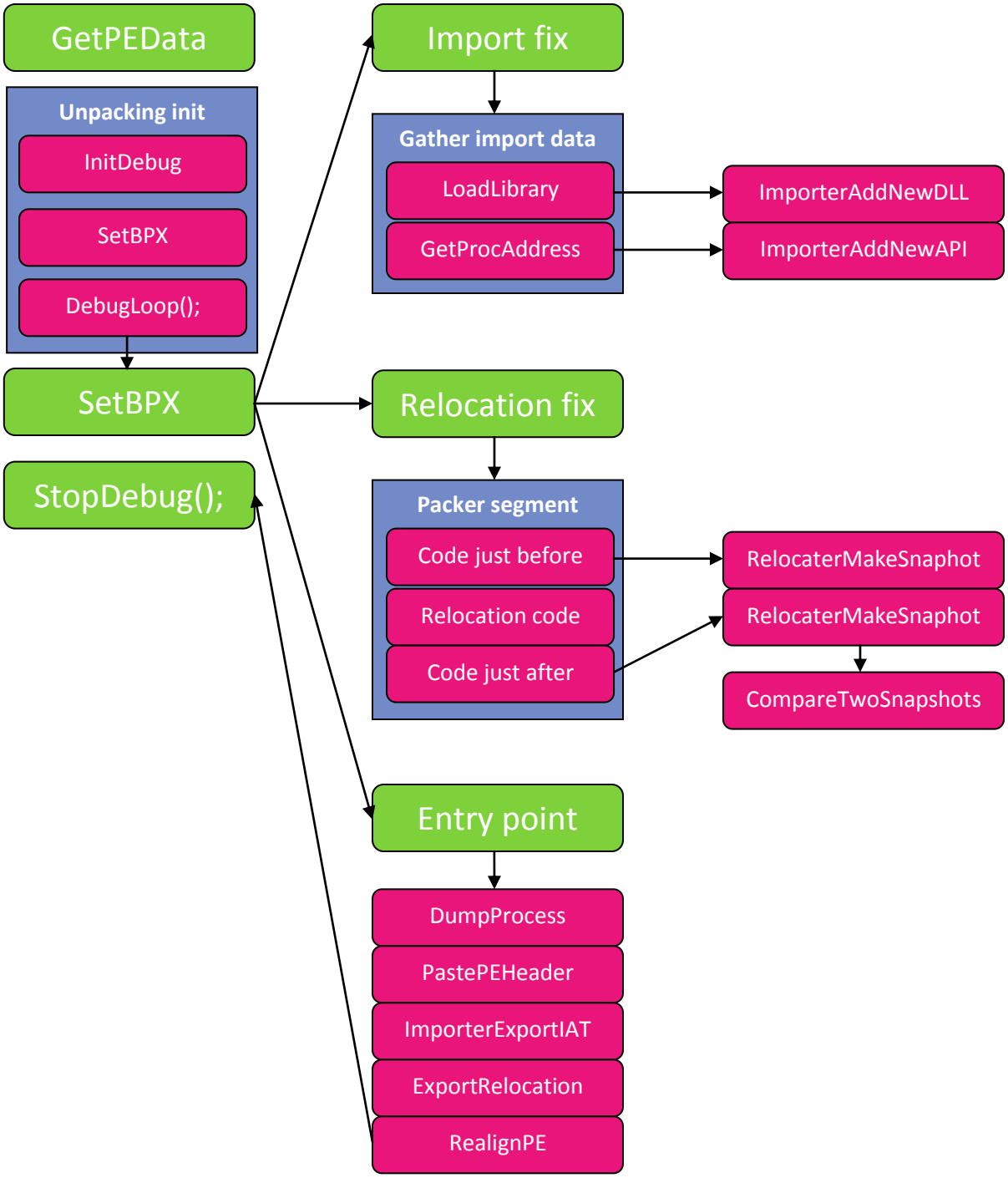
In order to use the *TitanEngine* SDK you must know the order in which APIs must be called. The order is pretty strict, and the layout of your unpacker will always be pretty much the same. Every unpacker starts with debugging and setting a breakpoint on your targets entry point. After this you must debug the program by running it until you get to the IAT filling code. This code uses LoadLibrary or GetModuleHandle API to load the dependent .dll files and GetProcAddress to find the locations of the necessary APIs. When this is done you need to break on original entry point, dump the file and paste imports to it.

To start the debugging you must first find the OEP address. To do this you can call GetPE32Data API and load the ImageBase and OriginalEntryPoint data. The sum of these two values is the address of the entry point. When this is done initialize the debugging by calling the InitDebug API. This API creates the debugged process but it does not start the actual debugging. In this suspended state call SetBpx to set the main breakpoint at OEP. This breakpoints callback will be called as soon as the debugged process finishes loading. To get the debugging process to this point you must call the DebugLoop API. After it is called the debugger takes over the debugging process. The only way to control the debugging process from this point is by callback procedures. Callbacks are set with all types of breakpoints. So if you set the breakpoint at OEP call the DebugLoop API first callback which will be called is the callback for original entry point breakpoint. Use that callback to set all the other breakpoints.

The best way to find where to set the breakpoint is by using the Find API. It will tell you the location on which your search pattern is found. If the packer for which you are writing an unpacker is single layered you can set all the breakpoints in the first or before the first callback, the one at original entry point. To get all the data needed to fix the IAT you need to set two or three breakpoints. First at LoadLibrary or GetModuleHandle API call, second at GetProcAddress call (use two if the packer calls GetProcAddress at two places for string API locating and ordinal API locating). But before you can actually call any of the importer functions in order to collect the IAT data you must first call ImporterInit API to initialize the importer.

After you do this and set the breakpoints you must code the breakpoint callbacks to get the IAT data. In LoadLibrary/GetModuleHandle callback you call ImporterAddNewDll API. One of its parameters is the string which holds the name of the .dll file which is loaded. This data location if located in a register or in a specific memory address. To get this data call the GetContextData API. If the data is a location on which the string is located and not an ordinal number you must call the ReadProcessMemory API to read the .dll name from the debugged processes. Then you can add the new .dll to importer engine. Note that once you add the .dll by calling the ImporterAddNewDll API all calls to ImporterAddNewAPI add APIs to last added .dll file. APIs are added by calling the ImporterAddNewAPI the same way you add a new .dll to importer engine. But unlike ImporterAddNewDll API here you must specify the location on which API pointer will be stored. This is a memory location on which pointer loaded with GetProcAddress API is written. After you collect all the data needed to fill in the IAT the unpacking is pretty much done.

Now you need to load the unpacked file OEP (this depends on the packer code) and dump the debugged process with DumpProcess. After this you can use the AddNewSection API to make space for the IAT which needs to be pasted to dump file. To know just how much space you need use the ImporterEstimatedSize API. Finally the IAT is exported to new section by calling the ImporterExportIAT API. Optionally you can realign the file and then you can stop the debugging process by calling the StopDebug API (if you don't stop the debugging target will keep running). This terminates the debugged process and then your program execution is resumed after the call to DebugLoop.



References and tools

[1] TitanEngine 2.0 SDK – www.reversinglabs.com

[2] Slides & Whitepaper: BlackHat, Vegas 2009 – blackhat.reversinglabs.com