The story of Binary Packers(Cryptors) and Kryptos

Murtaza Munaim

June 15 2013

1 Abstract

Kryptos is a packer which tries to make it harder than just one decrypt stub to be able to decrypt the packed code. What many packers do not utilize is the entire spectrum of transformations that a piece of code goes through before becoming binary code. The compiler is a powerful tool in giving information about the program, and this information can be used to create a much more powerful, and annoying packer. A basicblock is a collection of instructions that have no branching instructions except for at the end of the set of instructions which make up the basicblock. This is a useful unit of code to work with as it is guaranteed to continue execution one after another. The general idea behind Kryptos is that every function is a collection of basicblocks, which in turn are a set of opcodes(bytes). Thus a function is just a set of bytes with a label. Instead of encrypting all the executable code in the binary with one decryptor pattern stub to learn, each function has a unique decryptor stub and decryptor key. Further more, the decryptor key of the function is based and calculated only on successful execution of its calling function. An attacker on this system would have to either do expensive transition point guessing, which requires math intensive static analysis in order to emulate and solve the parameters of the packing code, or the attacker would have to actually run the program. Running a unknown program is always a challenge for an analyzer as resources are needed to setup a secure, sandboxed enviornment such as a virtual machine or a chroot(), because running unknown, unanalyzed, malicious code could have disastrous effects to the local enviornment if not taken into consideration.

Kryptos is designed to not add too much overhead to the program execution by using simple operations to pack the binary. The novelty in its design is that each function's key is reliant and derived on keys partially generated from previously called functions. It utilizes information from the compiler's abstract syntax tree in order to enumerate a callgraph of the code. This information is used to calculate which functions will calculate which keys. The mechanism of packing is twofold. One, the callgraph is traversed, and no-op instructions are inserted as markers and placeholders before and after each function. In code generation, optimizations are removed so no-ops are not removed in dead code analysis, and the compiler is allowed to create an executable binary with extra no-op instructions padding every user-defined function. This allows for a simple byte pattern to be searched for in the resulting binary. A tool is used to find the no-op padding and the function bytes inbetween and insert instead a decryptor stub inplace of the no-op instructions. The bytes of the function which were found will then be encrypted with the inverse of the mechanism determined by the decryptor stub. A set of varying decryptor stubs will be placed at each function. Examples of varying decryptor stubs are permutations of various common operations: add, sub, div, xor, mul, shr, shl. Ordering of instructions can be dynamically generated. The compiler framework LLVM is used to help generate callgraphs of code, while libbfd is used to rewrite the binary code in the resulting binary generated by the LLVM framework.

2 Introduction

2.1 Into to Packing

Binary packing is the modification of executable code and its binary file container, either by means of encryption, or compression.This modifies the opcodes and resulting file that was originally generated by the compiler for the target CPU architecture, and replaces them with an equivalent binary representation, the "packed" binary, defined by the binary packing designer. A packer/cryptor is the program that performs these binary modifications. The packer that "packs" the program will also insert equivalent instructions so that regular program execution can continue.

Code can be defined as X, the target CPU execution function defined as $C(T)$. With a packing function of $P(X, K = None)$, a cryptor would produce $Y = P(X, K)$ where Y is the resulting packed code. K is an optional key that might be used when the packing code has some encryption algorithm that requires a key. The equivalence $O_Y = C(Y) == O_X = C(X)$ must hold for the packer to be valid and work well. The definition of O might vary with what the packing designer wants. Key management, key derivation, deciding whether to use shared or public/private keys, for the packing function variable, K, is a seperate, and important issue to take care of with design.

2.2 Uses

Packing can be used for a multitude of uses. For example, Company A spent T amount of years developing algorithm X. Many resources were spent into the research and development, and thus Company A rightly wants to protect that investment. They still want to distribute their product to their customers. Distributing a packed library would allow their customers, and anyone who can get binary library, to use their code with some unpacking overhead. However, it would still make it hard for A's properietary algorithm to be ripped off, and relabeled as someone else's, as Company A can hide watermarking code within the packed library to provide proof that their originally made the library. Another reason for packing code is to make it hard for a binary to be easily disassembled. A dump of the opcodes of a packed binary would not produce valid instructions for the binary's target architecure. The ability to render disassemblers useless in analyzing a piece of binary code is vital in designing and shipping obfuscated code. Obfuscation is the goal in malicious code applications used to target client computers. Most anti-virus scanners use static heuristics in order to spot malicious code. This relies on the fact that virii and trojans will retain their original byte structure. This allows for byte signatutes to be developed for malicious code. Anti-virus engines can then use these byte patterns to spot certain bytes which translate to malicious function calls or known malicious data embedded within the virus payload. The use of binary packers defeats simple static byte signature engines as the same functional code can be packed in enumerable encodings, and finding and adding enumerable number of static byte siganatures is futile. It becomes a cat and mouse game, as most things are ;p. The cryptor is also a fun way to test reverse engineering skills in trying to decrypt and fully analyze the binary.

2.3 Contribution

One common way a packer or cryptor for the x86 CPU is implemented is as follows:

```
# x86 att syntax
decryptinit:
   xor eax, eax   # init key generation counter
   mov edi, 0x152A  # the number of integers in encrypted code
   mov ebp, 0x401000 # address of top of encrypted code
   mov esi, 0x44b3080 # seed key of encrypted code
loopbegin:
   mov ebx, [ebp] # read integer from key
   sub  ebx, esi # decrement key from integer
   xor ebx, esi # xor key from integer
   mov [ebp], ebx # store integer back where it came from
   inc eax # increment key counter
   add ebp, 4 # add 4 to the encrypted code index
   xor edx, edx # xor out edx for idiv
   mov ecx, 32 # 32 as modulo
   div ecx # eax = eax / ecx, edx = eax % ecx
   mov eax, edx # mov mod value into eax/ctr for shift
   rol  esi, eax # shift key left by the counter value circularly
   xor esi, edi # xor key by the counter of bytes processed
   dec edi # decrement number of integers to handle
   jne loopbegin # jump to loop begin to continue decrypting code
   jmp 0x401000 # jump to beginning to decrypted code
```

The problem I have seen with some binary packers that I have examined, and the example above, is that the instruction code is encrypted in one pass. This gives an attacker just one boundary to overcome in order to statically decrypt the binary. For example, the above code starts with a key of 0x44b3080, runs through 0x152a integers starting at address 0x401000. With some short code, one could emulate the above decryption code, and run it on the binary, effectively using static analysis to break the decryption. This can only be done, however, if the decryption method is known before hand.

2.4 Inspiration

There does, however, exist ways to detect a simple decryptor pattern. The code that runs before the final "jmp 0x401000" is called the decryptor stub. It is the only piece of code that has not been modified to be gibberish, unexecutable code, as the cryptor added the code to allow the binary to restore itself to its original, executable condition. A simple patterns occurs in most decryptor code, and that is the pattern of reading and writing from executable memory. This is a simple pattern that can be easily detected. These opcodes that are unencrypted, or, plaintext opcodes, can be considered transition points. First a suspicious binary can be identified by an entropy check.

Calculating the entropy of bytes in the binary is a good way to tell a binary might possibly be packed, as compressed and encrypted packing methods will change the average randomness of bytes compared to regular opcodes of the target processor. Then, a set of transition points, opcodes that match the pattern of load...store operations in short succession, followed by a unconditional jmp, all to addresses within the text section, can be identified. Once the points of "decryption" have been found, simple static analysis such as constant propogation(resolving which registers will recieve which constants based on solving some simple set of boolean relationships) and some opcode matching can identify the manner in which the bytes will be decrypted. These guessed decryptions can then be run on possible transition points. This would successfully decrypt a single-pass packed binary. [1]

2.5 Compiler Background

The compiler is a program that is used to turn a piece of human-understandable code into machine-understable code. There are many steps inbetween the input to output and each step allows for some code malleability. This could be either code optimizations/modifications to remove variables or loops with constants(constant propogation and loop unrolling, unfolding), code removal(dead code/variable analysis), or code additions(profiling code, anti-debug code, hidden messages, unpacker code). In this report we will be focusing on code additions by adding unpacker code with the aim to deter static code analysis of a binary, and force the analyzer to use dynamic analysis techniques to gather information about the program.

Source code is read in, an abstract syntax tree(AST) is generated from the code. The AST is a collection of graphs in a forest-like organization that represent the syntactic structure of the code. This is useful for performing optimizations and modifications to the code on the code such as dead code analysis, loop unrolling, constant propogation, adding extra wrapper code to a vital component piece of source code, parse the syntactic structure of the text, apply optimizations, and produce opcodes for the target machine that the code is supposed to run on be it x86, ARM, MIPS, etc. take a piece of source code, do some analysis, and produce and executable file.

3 Implemention

1. Create an llvm plugin/module/pass that will do two things. Use llvm to insert a set of nop instructions or some other sort of random binary pattern before the code generated for an function as well as markers describing the layout of the function in address order.
   (a) After code has been parsed, for all user-defined functions, a wrapper function will be made that takes in the same arguments as the original user-defined functions. This function will have some code that does the following:
      i. Fetches the address,len(in bytes), and key of the original function
      ii. Runs through a function. This function is taken from a .c file that is read in and used as the main packer. This function is used to to decrypt the address space of the original function.
      iii. Uses original function's passed arguments and jumps to original function's address
   (b) During Code generation for whichever architecture, a list of function addresses which have been marked "encrypt"? will be encrypted using its appropriate key and len of function. This lookup table should be the same as the one used by 1A.
2. use libbfd to find these pattern in the binary code to overwrite with the decryption code, and then use the function markers written by the ealier step to encrypt the function binary code, thus, an after-compile binary packer but using help from a compiler plugin.

4 Testing

Coming soon.

5 Conclusions

Need to implement more code ;p

References

[1] Kevin Patrick Coogan, *Deobfuscation of Packed and Virtualization-Obfuscation Protected Binaries*, The University of Arizona 2011