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 executable instructions that have no branching instructions except for at the end of 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. *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*

2 Introduction

2.1 Intro to Packing

Kryptos is designed to use a wide variety of information to pack a binary on a function-by-function basis. 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.

Binary packing is the modification of executable code, either by means of encryption, or compression. This modifies the opcodes and resulting file that was originally generated by the compiler and rewrites the file 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. Packing code also makes it harder to disassemble and statically analyze a binary. 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. Most malicious applications will use code obfuscation techniques when targetting client machines. 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. 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 with the binary packer example above, as well as others that I have examined, is that the code will unpack the executable code in just on epass. 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 unencrypted, 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).

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, parsing the syntactic structure of the text, apply optimizations, and producing 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

The implementation of this project is two fold, beginning with LLVM code modifications and additions, and ending with libbfd to manipulate the native opcodes, and insert the decryption meta information.

3.1 LLVM

The starting point of this project would be a piece of C source code that wants to be compiled to native code, and have appropriate packing/unpacking code added to the resulting code. This requires that a full compiler framework is built so that I could input code, manipulate it easily and without too much CPU-specific information. LLVM is a compiler framework library that bases most operations on its internal representation of code, which the library holds in an Intermediate Representation(IR) language. This IR language is a mix between low-level C code and assembly, but it is completely architecture and source code independent. This allows code to be handled in any which way as long as it has been converted to LLVM IR code. LLVM offers many programming language frontends so that a certain language can be "compiled" to LLVM IR. The library also allows for the internal IR code to be easily converted to a wide variety of native opcode sets for real machine execution. This allows for a source -¿ IR -¿ Modify -¿ IR -¿ native workflow to be made for almost anything that can be considered code. For our case, Clang is the C language frontend for LLVM used in this project.

We have two problems we want to address. First, we want to be able to adding packing code to the beginning of every function. Each function has a seperate key that is used for the decryption, as well as seperate unpacked code embedded as dominator nodes to the function. Dominator nodes are basicblocks that immedietley precede execution of the block that they are "dominating". For ease of design and to write trampoline unpacker code, the address of the function and the key used by the function will be embedded before each packer code. The trampoline packer code is then just required to pull back in memory from its current location to read function address and decryption key.

To make it easier for users of Kryptos, packer code is written in C code. Kryptos will compile the packer code to LLVM IR and insert into the analyzed C code. The user will also need to supply the inverse unpacker code for each packer function the user inputs into the Kryptos packer C code. This requires Kryptos to have 2 C files as input, one source file to be packed, one source with packing and unpacking code. The packing code C file will contain multiple unpacker functions that the Kryptos user wants to have used in the packer as unpacker stubs. Each packer stub will also have a unique ID so that the inverse packer code can be identifed for use later in the libbfd stage of Kryptos. Below is the pseudocode for the operation of the LLVM portion of Kryptos

1. Read topack.c and packers.c (code to pack, and packer code, respectively)
2. Compile topack.c and packers.c to topack.IR and packers.IR
3. For currfunc in topack.IR:
   (a) Take one random function from packers.IR as currpack.IR
   (b) Create empty dominator node and add to beginning of currfunc, adding edge from node to first node of currfunc
   (c) Add PADCOUNT * 1000 NOPS to the beginning of the function
   (d) Take currfunc.IR's packer function IR and insert db funcid like instruction, so that the first byte after the nop identifies packer
   (e) Add instructions to grab key bytes for currpack.IR function from currloc - len(KEYLEN=16)
   (f) Add instructions to grab function address bytes for currpack.IR function from currloc - len(KEYLEN=16) - archwidth
   (g) Take currpack.IR and add to the top node of currfunc.
   (h) Add PADCOUNT * 1000 NOPS to the end of the function
   (i) Verify function has well-formed basicblocks with llvmpy.core.Module.verify()
4. Turn off all optimizations so pad nops are not removed
5. Generate native binary from modified internal LLVM IR

3.2 libbfd

libbfd stands for Binary File Descriptor library. It is a useful library to do post-compiled binary modifications to any type of binary generated from some sort of compiler. In this case, we use libbfd to modify the bytes from the resulting LLVM-generated binary. The number of PAD bytes added in the previous step allows us to look for a repeated set of bytes that we know for a fact will be at a certain count, with optimizations turned off. This aids us to find where in the binary we need to make binary modifications. Also, because space is reserved for the function address, and decryption key, this libbfd pass of Kryptos can use any key independent of the LLVM code insertion process, and use the compiler-calculated function addresses to insert into the reserved binary space so that the instrumented LLVM instructions can read the address at runtime. Below is the psuedo-code of the libbfd pass of Kryptos:

1. For match in findall(binaryfile, PADCOUNT * 1000 NOPS):

(a) Grab packer identifer code which lies after the PADCOUNT to find which inversed packer code libbfd needs to call.

(b) Find the address of the location after the packer identifier byte. This is the first instruction in the code to be packed

(c) Using identifed packer code along with parameters of function address calculated above, generated random key(if not user defined), run the packing code until the PADCOUNT * 1000 NOPS function end marker has been found by the string searcher.

4 Testing

The testing of this project was simple. First, proper functionality of the code to be able to add arbitrary code was tested. After being able to add in any instrumentation code I wanted, I verified the added instructions using objdump -d to see the resulting native opcodes. Ssome sample packer/unpacker stubs were written in C so that I could insert real packing code into arbitrary C files. Code generation and execution of the IR-¿native code was done on every iteration so that original functionality of the program to the packed wasn't modified too much.

5 Conclusions

Utilizing a mix of mid-compilation and post-compilation data helps to create a packer tool that is a bit smarter than the average malware. Packing executable code on a function by funtion level makes the problem of statically analyzing malware alot harder, and requires a dynamic analysis approach, which in most cases is undesired. Dynamic analysis requires live analysis and when running malware, predicting what the code will do that is potentially dangerous is alot better then just making a safe enviornment and letting it go boom. On a case by case basis, this protocol could be reverse engineered, for the basis on packer insertion is relatively simple. It can be made a powerful by the choice of packer/unpacker stubs that the user of Kryptos chooses to write, be it an easy xor encryption stub or maybe heavy-weight AES with some anti-debug tricks included.

Code and test files are available at: https://bitbucket.org/manizzle/kryptos

References

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