A Case Study into solving Crypters/Packers in Malware Obfuscation using an SMT approach

Jason Reaves

ABSTRACT
Obfuscation in malware is commonly employed for any number of reasons but its purpose is ultimately the same, to make the underlying malicious entity go unnoticed. Crypters and Packers both are heavily employed to bypass common security measures so ultimately these are just tools. Tools that are utilizing algorithms in order to take data and turn it into some other data while being able to reverse the process later, obviously these reversible algorithms can be chained together as well into ‘layers’. In this paper I explore the idea that it is easier to think of these layers as a math equation which can be solved. This has the potential of turning something that can be overwhelming at first, like writing an unpacker, into a much more manageable problem.

For the purpose of this paper I will refer to packers[9] and crypters[9] both as packers, the reason being that in the world of malware both are used for obfuscating the underlying code that is to be executed.

1. Introduction
Packers have evolved greatly over the years, especially with malware needing to utilize crypters and packers that can bypass any number of obstacles depending on their targets. For brevity we will focus specifically on crypters that utilize multiple binary operators to obfuscate their payloads, it seems a natural progression that researchers will usually move towards finding ways to pivot from other data in these scenarios such as finding ways to rip out starting values through various techniques including bruting, select-bruting, regex matching, nearby static data pivoting or any number of other process for basically finding values. Instead of finding values I always yearned to be able to instead lean on math in regards to solving a problem, if I can reverse this routine and describe it in an adequate manner to write it in a higher level language then I should be able to describe this routine as a problem that can either be simplified or best case
solved. This line of thought is what eventually led me to find Z3[6] and its usefulness in subsets of malware research.

2. Finding the problem
The sample we’ll be looking at is specifically the crypter being used by the latest Locky Ransomware campaigns in late August 2017, 1c80b1ba2c514bc1d32eb5b9909d79812ab8f2944548bc96757c1d992ce6d8ac. While the object of this paper is not to show how to reverse engineer routines or malware, we will simply walk through to the relevant portion of code in order to begin describing our problem. Basically we’re going to find where the routine that is responsible for decoding the payload. For this crypter a quick glance at the PE file shows a potentially encoded resource section.

Opening up the file in a debugger shows a bunch of very similar calls at the main entrypoint[10].

![Resource sections](image)
Peeking inside one of these calls shows that they are just jump commands to OpenMutex. So it is trying to open a mutex with the desired access of SYNCHRONIZE[8].
As long as all the calls return 0 then the code will come to a different function call that takes us to a different section of the binary that starts to make some LoadLibrary calls.

A quick stop at a loop that calls WaitForSingleObject over and over, potentially a custom sleep routine. Sleep routines are commonly leveraged in malware to defeat sandbox analysis which will normally only execute a piece of malware for a set time amount[11].
Moving on a little later we see a call to VirtualAlloc followed by a loop utilizing a push->ret technique. Unfortunately this isn’t our routine for decoding the payload but instead the routine for decoding the bytecode layer that will be called next[12], do we need this layer? Possibly, whether or not we need to decode out that layer will depend on how the final routine is implemented for decoding out the payload. If you’d like an example of a slightly more advanced example of a crypter where we end up having to decode out some of the layers of a crypter I have a write up on one such crypter[1] where the decoding routine is dynamically generated and needs to be decoded.

![Figure 6 Decode and execute next layer](image)

Heading into that next layer is just your normal code resolving any dependencies that it needs at runtime[13].
You might notice with this next picture the address change, simply because the bytecode layer fixes its own dependencies and then allocates a new memory section and copies itself over before calling the next section of code to be executed from within itself, kind of an odd way to do it but if you’re the type that sets breakpoints everywhere you might find yourself with messed up code. In this next code however we have a call to VirtualAlloc followed by some data being moved into our newly created memory.
Whenever I see something like this in a crypter the first thought that comes to my mind is “where is this data located in the binary”. A quick check shows it’s the resource section we had noticed when we were doing our precursory inspection.
The next call after the data is moved is interesting, some hardcoded dword values, two sub instructions with a load and store in a loop? That looks like an encoding loop of some kind.

![Figure 10 Decoding routine](image)

It’s good in these situations to keep track of what and where any hardcoded values are, such as the one loaded into EDX immediately and then added to a hardcoded value, turns out both values are hardcoded in the bytecode layer.

![Figure 11 Hardcoded value](image)
Further down we see the previously mentioned loop that if you step through a few times you’ll notice the PE file emerge, so this is the loop that we are concerned with since we know the file is in a resource section for this particular sample.

![Figure 12 Decoding values](image)

We have one hardcoded value and the previous two hardcoded values added together to get us the first two values subtracted, afterwards you can see EDX which contained one of those values is replaced with the previous dword value from our encoded data. We can construct this as a math routine:

\[
f(x) = x - 0x824132AC - \Delta
\]

![Figure 13 Proposed initial function](image)

We know that delta is 0x3c662605 for the first iteration and delta becomes the previous x as it loops over the data. However when we are looking to decode out the binary we won’t know the hardcoded value 0x824132ac and we also won’t know the starting delta value. Simple enough to think of bruting out the values but that could be a pain, you would need to brute out one value from what you would expect to see in the first four bytes of a PE file and then try to figure out what the hardcoded value is from the next 4 bytes. Possible but could take a few cycles to brute, so instead you could decode out the bytecode layer and then use YARA[7] and regex patterns to try to find possible values instead to simplify this process but this approach can be error prone and end up being just as slow as bruting depending on how you implement it. The other option is to use an SMT solver, they can solve these types of problems very quickly because we know the endgame is a PE file and a PE file has a bunch of header data that we can predict.

3. SMT solving an unpacker
We basically did a walkthrough of the routine that decodes out the payload in the previous section. Up next we’re going to go through how to turn this decoding routine, which is basically a math problem, into something that can be solved by an SMT.
Since we know the encoded data is in a resource section we can setup our overall program pseudocode as thus:

```python
unpacked = None
rsrcs = get_resource_sections(data)
for rsr in rsrscs:
    s = smt_solve(rsr)
    if s.solved():
        sub1 = s['sub_value_1']
        sub2 = s['sub_value_2']
        unpacked = decode(rsr, sub1, sub2)
        break
```

The gist is we will call a function on every resource section which will handle setting up our SMT solver by adding in necessary constraints. What are our constraints? Simply that we know what the output of the decoding should be, a PE file, and we know the routine involved. This means the process of setting up our solver and adding constraints is basically just describing a problem and then letting it solve the problem for us.

Ultimately we have two values that we need to find, a hardcoded subtract DWORD value and another DWORD value that only gets used for the first iteration and then replaced with the previous encoded DWORD value. For our problem these values basically become variables and for simplicity we can use the first 12 bytes of the unpacked PE file of the sample we just went through, '\x4d\x5a\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00'. If you’re asking “why 12 bytes”, the answer is basically that the more data you have the more likely you are able to solve the problem for the actual variables. If you only add a constraint for the first 4 bytes, given that the values we are trying to find are DWORD values this gives us many possibilities to satisfy the problem. The more data we can add for constraints then the narrower we can make the list of possible values to satisfy our problem which in turn gives us a better chance of producing the actual values we need instead of a possible range of values. Let’s begin setting up our solver function.
def solve_doublesub(input, output):
    hc_sub = BitVec('sub1', 32)
    delta_sub = BitVec('sub2', 32)
    s = Solver()  

Here we’ve setup the beginning of our solver function and declared them as BitVecs which are basically variables, in this case 32 bit variables named sub1 and sub2 respectively. These will represent our hardcoded subtraction variable and our initial delta variable that we are trying to find out. The code can seem a bit weird at first, I found it best to think of these are your variable declarations. How you use your variables is by taking our math function above and unrolling a few iterations of the function into their equivalent y=x version which let’s us find the unknown values we are searching for because we know the y and x or the output and the input. Let’s take another look at our function.

\[ f(x) = x - 0x824132AC - \Delta \]

Now let’s replace some of the data to make it use our variables and turn it into the y=x form.

\[ y = x - hcsub - deltasub \]

We mentioned earlier that we know the inputs and the outputs already, the inputs are bytes that can be found inside our sample. The outputs are the first few bytes of a normal windows executable(\x4d\x5a\x90\x00\x03\x00\x04\x00\x00\x00\x00\x00\x00\x00'), with this along with the inputs which as aforementioned are bytes that can be found inside the malware sample we are able to use our mathematical algorithm to find the values we need. We set this up by adding constraints which are conditions that will constraint our SMT solver.
second_delta = struct.unpack_from('<I', input)[0]
s.add((BitVecVal(struct.unpack_from('<I', input), 32) - hc_sub) - delta_sub
   == BitVecVal(struct.unpack_from('<I', output), 32))
s.add((BitVecVal(struct.unpack_from('<I', input[4:]), 32) - hc_sub) -
   second_delta == BitVecVal(struct.unpack_from('<I', output[4:]), 32))
s.add((BitVecVal(struct.unpack_from('<I', input[8:]), 32) - hc_sub) -
   BitVecVal(struct.unpack_from('<I', output[8:]), 32)) ==
   BitVecVal(struct.unpack_from('<I', output[8:]), 32))
return(s)

Figure 18 Setup SMT constraints

Now we can loop through every resource section and look for one that satisfies our constraints in the solver.

for rsrcc in rsrccs:
    #Try z3 solvers
    a = bytearray(rsrcc)
    for poss_decode in possible_decodes:
        s = solve_doublesub(a, poss_decode)
        if s.check() == sat:
            m = s.model()
            for d in m.decls():
                if d.name() == 'sub1':
                    sub1 = m[d].as_long()
                elif d.name() == 'sub2':
                    sub2 = m[d].as_long()
            print("Satisfied!")
            print("Sub1 Value: "+hex(sub1))
            print("Sub2 Value: "+hex(sub2))

Figure 19 Try to find the encoded file
4. Conclusion and Future Work

In this paper I detailed a concept on how to approach looking at packers but creating an unpacker in this manner is not something feasible for every variant that exists, it is possible to do in one off scenarios where a researcher is tracking a specific malware family using a packer and wants to find what other families might either be used by the same group(for example a private packer or not sold) or perhaps what other groups are using the same packer(for example a public packer that is sold as a service). Either discovery tells a different story that can be useful for a researcher trying to paint a better picture over the workings of a threat group that might be leveraging malware[1,2]

There are some ways that current software could leverage some of the concepts presented in this paper however, auto unpacking solutions have existed for a number of years and they normally rely on a combination of sandboxes or virtual machines with specific loaded modules or software designed to look for binaries that are decoded and rebuilt into memory sections[3,4,5]. The concept here specifically of leveraging the decoding routines themselves could be used to expand the usefulness of these existing automated systems for finding interesting code sections that might not be detected via normal means. The main idea being that if the malware is decoding something then it’s potentially useful to someone so if you can find that specific routine you can harvest everything it decodes in an automated manner without having to guess later what was or wasn’t decoded. Such a system would need a way to heuristically detect where these routines are, a way to detect when they are executed and a way to dump or store the decoded data after they have finished running.

It might also be beneficial for automated systems to detect signatures for the decoded data and then in the event of a miss for the decode routine but a detection on a decoded data signature determining in the execution where this signature fired so the address of the routine can be stored and potentially the entire execution restarted while monitoring this routine to harvest all decoded data.
This allows the possibility of harvesting more data from malware in an automated fashion but also being able to determine the most interesting routines to harvest which are the ones responsible for decoding data, these routines could then be passed to an engine designed for auto generating detecting binary detection rules or also stored for further review by researchers/analysts.

Future research will involve the heuristic static detection of malware and interesting routines that this concept could be leveraged against and a process of auto generating SMT solvers.

5. References

3: https://cuckoosandbox.org/
4: https://www.unpac.me/
5: https://github.com/hasherezade/pe-sieve
6: https://github.com/Z3Prover/z3
7: https://virustotal.github.io/yara/
8: https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw
11: https://attack.mitre.org/techniques/T1497/
12: https://attack.mitre.org/techniques/T1406/