Deobfuscating APT32 Flow Graphs with Cutter and Radare2

By deugenio

Published: 2019-04-24 · Archived: 2026-04-10 02:57:08 UTC

Research by: Itay Cohen

The Ocean Lotus group, also known as APT32, is a threat actor which has been known to target East Asian countries such as Vietnam, Laos and the Philippines. The group strongly focuses on Vietnam, especially private sector companies that are investing in a wide variety of industrial sectors in the country. While private sector companies are the group's main targets, APT32 has also been known to target foreign governments, dissidents, activists, and journalists.

APT32's toolset is wide and varied. It contains both advanced and simple components; it is a mixture of handcrafted tools and commercial or open-source ones, such as Mimikatz and Cobalt Strike. It runs the gamut from droppers, shellcode snippets, through decoy documents and backdoors. Many of these tools are highly obfuscated and seasoned, augmented with different techniques to make them harder to reverse-engineer.

In this article, we get up and close with one of these obfuscation techniques. This specific technique was used in a backdoor of Ocean Lotus' tool collection. We'll describe the technique and the difficulty it presents to analysts — and then show how bypassing this kind of technique is a matter of writing a simple script, as long as you know what you are doing.

The deobfuscation plugin requires Cutter, the official GUI of the open-source reverse engineering framework – radare2. Cutter is a cross-platform GUI that aims to expose radare2's functionality as a user-friendly and modern interface. Last month, Cutter introduced a new Python plugin system, which figures into the tool we'll be constructing below. The plugin itself isn't complicated, and neither is the solution we demonstrate below. If simple works, then simple is best.

Downloading and installing Cutter

Cutter is available for all platforms (Linux, OS X, Windows). You can download the latest release here. If you are using Linux, the fastest way to get a working copy of Cutter is to use the AppImage file.

If you want to use the newest version available, with new features and bug fixes, you should build Cutter from source. If you are up for that detour, follow this tutorial.

Fig 1: *Cutter interface*

The Backdoor

First, let's have a look at the backdoor itself. The relevant sample (`486be6b1ec73d98fdd3999abe2fa04368933a2ec`) is part of a multi-stage infection chain, which we have lately seen employed in the wild. All these stages are quite typical for Ocean Lotus, especially the chain origin being a malicious document (`115f3cb5bdfb2ffe5168ecb36b9aed54`). The document purports to originate from Chinese security vendor Qihoo 360, and contains a malicious VBA Macro code that injects a malicious shellcode to `rundll32.exe.` The shellcode contains decryption routines to decrypt and reflectively load a DLL file to the memory. The DLL contains the backdoor logic itself.

First, the backdoor decrypts a configuration file which is pulled from the file resource. The configuration file stores information such as the Command and Control servers. The binary then tries to load an auxiliary DLL to the memory using a custom-made PE loader. This DLL is called `HTTPProv.dll` and is capable of communicating with the C2 servers. The backdoor can receive dozens of different commands from the Command and Control servers, including shellcode execution, creation of new processes, manipulation of files and directories, and more.

Many obfuscation techniques are used by Ocean Lotus in order to make their tools harder to reverse engineer. Most noticeable, Ocean Lotus is using an enormous amount of junk code in their binaries. The junk code makes the samples much bigger and more complicated, which distracts researchers trying to pry into the binary. Trying to decompile some of these obfuscated functions is a lost cause; the assembly often plays around with the stack pointer, and decompilers are not well-equipped to handle this kind of pathological code.

The Obfuscation

Upon analysis of the backdoor, one obfuscation technique can be immediately noticed. It is the heavy use of control flow obfuscation which is created by inserting junk blocks into the flow of the function. These junk blocks

are just meaningless noise and make the flow of the function confusing.

Fig 2: *An example of a junk block*

As you can see in the image above, the block is full of junk code which has nothing to do with what the function actually does. It's best to ignore these blocks, but that's easier said than done. A closer look at these blocks will reveal something interesting. These junk blocks are always being fail-jumped to by a conditional jump from a previous block. Furthermore, these junk blocks will almost always end with a conditional jump which is the opposite of the conditional jump of the previous block. For example, if the condition above the junk block was `jo`

`<some_addr>`, the junk block will most likely end with `jno <some_addr>`. If the block above ended with `jne <another_addr>`, the junk block will then end with… you guessed right – `je <another_addr>`.

Fig 3: *Opposite conditional jumps*

With this in mind, we can begin structuring the characteristics of these junk blocks. The first characteristic of the obfuscation is the **occurrence of two successive blocks which end with opposite conditional jumps to the same target address**. The other characteristic requires the **second block to contain no meaningful instructions such as string references or calls**.

When these two characteristics are met, we can say with a high chance that the second block is a junk block. In such a case, we would want the first block to jump over the junk block so the junk block would be removed from the graph. This can be done by patching the conditional jump with an unconditional jump, aka a simple `JMP` instruction.

Fig 4: *Modifying the conditional jump to a JMP instruction will ignore the junk block*

Writing the Plugin

So here is a heads up for you – the plugin we present below is written for Cutter, but was designed to be compatible with radare2 scripts, for those of you who are CLI gurus. That means that we are going to use some nifty radare2 commands through r2pipe – a Python wrapper to interact with radare2. This is the most effective and flexible way for scripting radare2.

It's not trivial to get the plugin to support both Cutter and radare2, since one is a GUI program and the other is a CLI. That means that GUI objects would be meaningless inside radare2. Luckily, Cutter supports r2pipe and is able to execute radare2 commands from inside its Python plugins.

Writing the Core Class

The first thing we are going to do is to create a Python class which will be our core class. This class will contain our logic for finding and removing the junk blocks. Let's start by defining its `__init__` function. The function will receive a pipe, which will be either an `r2pipe` (available from `import r2pipe`) object from radare2 or a `cutter` (available from `import cutter`) object from Cutter.

```
class GraphDeobfuscator:
   def __init__(self, pipe):
       """an initialization function for the class

       Arguments:
           pipe {r2pipe} -- an instance of r2pipe or Cutter's wrapper
       """

       self.pipe = pipe
```

Now we can execute radare2 commands using this pipe. The pipe object contains two major ways to execute r2 commands. The first is `pipe.cmd(<command>)` which will return the results of the command as a string, and the second is `pipe.cmdj(<command>j)` which will return a parsed JSON object from the output of radare2's command.

> **Note:** *Almost every command of radare2 can be appended with a `j` to get the output as JSON.*

The next thing we would want to do is to get all the blocks of the current function and then iterate over each one of them. We can do this by using the `afbj` command which stands for **A**nalyze **F**unction **B**locks and will return a **J**son object with all the blocks of the function.

```
   def clean_junk_blocks(self):
       """Search a given function for junk blocks, remove them and fix the flow.
       """
       # Get all the basic blocks of the function
       blocks = self.pipe.cmdj("afbj @ $F")
       if not blocks:
           print("[X] No blocks found. Is it a function?")
           return
       modified = False

       # Iterate over all the basic blocks of the function
       for block in blocks:
           # do something
```

For each block, we want to know if there is a block which fails-to in a case where the conditional jump would not take place. If a block has a block to which it fails, the second block is an initial candidate to be a junk block.

```
   def get_fail_block(self, block):
       """Return the block to which a block branches if the condition is fails

       Arguments:
           block {block_context} -- A JSON representation of a block

       Returns:
```

```
            block_context -- The block to which the branch fails. If not exists, returns None
        """
        # Get the address of the "fail" branch
        fail_addr = self.get_fail(block)
        if not fail_addr:
            return None
        # Get a block context of the fail address
        fail_block = self.get_block(fail_addr)
        return fail_block if fail_block else None
```

> **Note:** *Since our space is limited, we won't explain every function that appears here. Functions as* `get_block (addr)` *or* `get_fail_addr (block)` *that are used in the snippet above are subroutines we wrote to make the code cleaner. The function implementations will be available in the final plugin that is shown and linked at the end of the article. Hopefully, you'll find the function names self-explanatory.*

Next, we would like to check whether our junk block candidate comes immediately after the block. If no, this is most likely not a junk block since from what we inspected, junk blocks are located in the code immediately after the blocks with the conditional jump.

```
    def is_successive_fail(self, block_A, block_B):
        """Check if the end address of block_A is the start of block_B

        Arguments:
            block_A {block_context} -- A JSON object to represent the first block
            block_B {block_context} -- A JSON object to represent the second block

        Returns:
            bool -- True if block_B comes immediately after block_A, False otherwise
        """

       return ((block_A["addr"] + block_A["size"]) == block_B["addr"])
```

Then, we would want to check whether the block candidate contains no meaningful instructions. For example, it is unlikely that a junk block will contain `CALL` instructions or references for strings. To do this, we will use the command `pdsb` which stands for **P**rint **D**isassembly **S**ummary of a **B**lock. This radare2 command prints the interesting instructions that appear in a certain block. We assume that a junk block would not contain interesting instructions.

```
    def contains_meaningful_instructions (self, block):
        '''Check if a block contains meaningful instructions (references, calls, strings,...)

        Arguments:
            block {block_context} -- A JSON object which represents a block

        Returns:
```

```
            bool -- True if the block contains meaningful instructions, False otherwise
        '''

        # Get summary of block - strings, calls, references
        summary = self.pipe.cmd("pdsb @ {addr}".format(addr=block["addr"]))
        return summary != ""
```

Last, we would like to check whether the conditional jumps of both blocks are opposite. This will be the last piece of the puzzle to determine whether we are dealing with a junk block. For this, we would need to create a list of opposite conditional jumps. The list we'll show is partial since the x86 architecture contains many conditional jump instructions. That said, from our tests, it seems like the below list is enough to cover all the different pairs of opposite conditional jumps that are presented in APT32's backdoor. If it doesn't, adding additional instructions is easy.

```
    jmp_pairs = [
        ['jno', 'jo'],
        ['jnp', 'jp'],
        ['jb',  'jnb'],
        ['jl',  'jnl'],
        ['je',  'jne'],
        ['jns', 'js'],
        ['jnz', 'jz'],
        ['jc',  'jnc'],
        ['ja', 'jbe'],
        ['jae', 'jb'],
        ['je',  'jnz'],
        ['jg',  'jle'],
        ['jge', 'jl'],
        ['jpe', 'jpo'],
        ['jne', 'jz']]

    def is_opposite_conditional(self, cond_A, cond_B):
        """Check if two operands are opposite conditional jump operands

        Arguments:
            cond_A {string} -- the conditional jump operand of the first block
            cond_B {string} -- the conditional jump operand of the second block

        Returns:
            bool -- True if the operands are opposite, False otherwise
        """

        sorted_pair = sorted([cond_A, cond_B])
        for pair in self.jmp_pairs:
            if sorted_pair == pair:
```

```
                return True
        return False
```

Now that we defined the validation functions, we can glue these parts together inside the `clean_junk_blocks()` function we created earlier.

```
    def clean_junk_blocks(self):
        """Search a given function for junk blocks, remove them and fix the flow.
        """

        # Get all the basic blocks of the function
        blocks = self.pipe.cmdj("afbj @ $F")
        if not blocks:
            print("[X] No blocks found. Is it a function?")
            return
        modified = False

        # Iterate over all the basic blocks of the function
        for block in blocks:
            fail_block = self.get_fail_block(block)
            if not fail_block or \
            not self.is_successive_fail(block, fail_block) or \
            self.contains_meaningful_instructions(fail_block) or \
            not self.is_opposite_conditional(self.get_last_mnem_of_block(block), self.get_last_mnem_o
                continue
```

In case that all the checks are successfully passed, and we can say with a high chance that we found a junk block, we would want to patch the first conditional jump to `JMP` over the junk block, hence removing the junk block from the graph and thus, from the function itself.

To do this, we use two radare2 commands. The first is `aoj @ <addr>` which stands for **A**nalyze **O**pcode and will give us information on the instruction in a given address. This command can be used to get the target address of the conditional jump. The second command we use is `wai <instruction> @ <addr>` which stands for **W**rite **A**ssembly **I**nside. Unlike the `wa <instruction> @ <addr>` command to overwrite an instruction with another instruction, the `wai` command will fill the remaining bytes with `NOP` instructions. Thus, in a case where the `JMP <addr>` instruction we want to use is shorter than the current conditional-jump instruction, the remaining bytes will be replaced with `NOP`s.

```
    def overwrite_instruction(self, addr):
        """Overwrite a conditional jump to an address, with a JMP to it

        Arguments:
            addr {addr} -- address of an instruction to be overwritten
        """
```

```
        jump_destination = self.get_jump(self.pipe.cmdj("aoj @ {addr}".format(addr=addr))[0])
        if (jump_destination):
            self.pipe.cmd("wai jmp 0x{dest:x} @ {addr}".format(dest=jump_destination, addr=addr))
```

After overwriting the conditional-jump instruction, we continue to loop over all the blocks of the function and repeat the steps described above. Last, if changes were made in the function, we re-analyze the function so that the changes we made appear in the function graph.

```
    def reanalize_function(self):
        """Re-Analyze a function at a given address

        Arguments:
            addr {addr} -- an address of a function to be re-analyze
        """
        # Seek to the function's start
        self.pipe.cmd("s $F")
        # Undefine the function in this address
        self.pipe.cmd("af- $")

        # Define and analyze a function in this address
        self.pipe.cmd("afr @ $")
```

At last, the `clean_junk_blocks()` function is now ready to be used. We can now also create a function, `clean_graph()`, that cleans the obfuscated function of the backdoor.

```
    def clean_junk_blocks(self):
        """Search a given function for junk blocks, remove them and fix the flow.
        """

        # Get all the basic blocks of the function
        blocks = self.pipe.cmdj("afbj @ $F")
        if not blocks:
            print("[X] No blocks found. Is it a function?")
            return
        # Have we modified any instruction in the function?
        # If so, a reanalyze of the function is required
        modified = False

        # Iterate over all the basic blocks of the function
        for block in blocks:
            fail_block = self.get_fail_block(block)
            # Make validation checks
            if not fail_block or \
            not self.is_successive_fail(block, fail_block) or \
            self.contains_meaningful_instructions(fail_block) or \
```

```
            not self.is_opposite_conditional(self.get_last_mnem_of_block(block), self.get_last_mnem_o
                continue
            self.overwrite_instruction(self.get_block_end(block))
            modified = True
        if modified:
            self.reanalize_function()


    def clean_graph(self):
        """the initial function of the class. Responsible to enable cache and start the cleaning
        """

        # Enable cache writing mode. changes will only take place in the session and
        # will not override the binary
        self.pipe.cmd("e io.cache=true")
        self.clean_junk_blocks()
```

This concludes the core class.

Cutter or Radare2?

As mentioned earlier, our code will be executed either as a plugin for Cutter, or straight from the radare2 CLI as a Python script. That means that we need to have a way to understand whether our code is being executed from Cutter or from radare2. For this, we can use the following simple trick.

```
# Check if we're running from cutter
try:
   import cutter
   from PySide2.QtWidgets import QAction
   pipe = cutter
   cutter_available = True
# If no, assume running from radare2
except:
   import r2pipe
   pipe = r2pipe.open()
   cutter_available = False
```

The code above checks whether the `cutter` library can be imported. If it can, we are running from inside Cutter and can feel safe to do some GUI magic. Otherwise, we're running from inside radare2, and so we opt to import `r2pipe`. In both statements, we are assigning a variable named `pipe` which will be later passed to the `GraphDeobfuscator` class we created.

Running from Radare2

This is the simplest way to use this plugin. Checking if `__name__` equals "__main__" is a common Python idiom that checks if the script was run directly or imported. If this script was run directly, we simply execute the

`clean_graph()` function.

```
if __name__ == "__main__":
   graph_deobfuscator = GraphDeobfuscator(pipe)
   graph_deobfuscator.clean_graph()
```

Running from Cutter

Cutter's documentation describes how to go about building and executing a Plugin for Cutter, and we follow its lead. First, we need to make sure that we are running from inside Cutter. We already created a boolean variable named `cutter_variable`. We simply need to check whether this variable is set to `True`. If it does, we proceed to define our plugin class.

```
if cutter_available:
   # This part will be executed only if Cutter is available.
   # This will create the cutter plugin and UI objects for the plugin
   class GraphDeobfuscatorCutter(cutter.CutterPlugin):
       name = "APT32 Graph Deobfuscator"
       description = "Graph Deobfuscator for APT32 Samples"
       version = "1.0"
       author = "Itay Cohen (@Megabeets_)"

       def setupPlugin(self):
           pass

       def setupInterface(self, main):
           pass

   def create_cutter_plugin():
       return GraphDeobfuscatorCutter()
```

This is a skeleton of a Cutter plugin — it contains no proper functionality at all. The function `create_cutter_plugin()` is called by Cutter upon loading. At this point, if we will place our script in Cutter's plugins directory, Cutter will recognize our file as a plugin.

To make the plugin execute our functionality, we need to add a menu entry that the user can press to trigger our deobfuscator. We chose to add a menu entry, or an Action, to the "***Windows -> Plugins***" menu.

```
if cutter_available:
   # This part will be executed only if Cutter is available. This will
   # create the cutter plugin and UI objects for the plugin
   class GraphDeobfuscatorCutter(cutter.CutterPlugin):
       name = "APT32 Graph Deobfuscator"
       description = "Graph Deobfuscator for APT32 Samples"
       version = "1.0"
```

```
        author = "Megabeets"

        def setupPlugin(self):
            pass

        def setupInterface(self, main):
            # Create a new action (menu item)
            action = QAction("APT32 Graph Deobfuscator", main)
            action.setCheckable(False)
            # Connect the action to a function - cleaner.
            # A click on this action will trigger the function
            action.triggered.connect(self.cleaner)

            # Add the action to the "Windows -> Plugins" menu
            pluginsMenu = main.getMenuByType(main.MenuType.Plugins)
            pluginsMenu.addAction(action)

        def cleaner(self):
            graph_deobfuscator = GraphDeobfuscator(pipe)
            graph_deobfuscator.clean_graph()
            cutter.refresh()


    def create_cutter_plugin():
        return GraphDeobfuscatorCutter()
```

The script is now ready, and can be placed in the Python folder, under Cutter's plugins directory. The path to the directory is shown in the Plugins Options, under "***Edit -> Preferences -> Plugins***". For example, on our machine the path is: "*~/.local/share/RadareOrg/Cutter/Plugins/Python*".

Now, when opening Cutter, we can see in "***Plugins -> Preferences***" that the plugin was indeed loaded.

Fig 5: *The plugin was successfully loaded*

We can also check the "***Windows -> Plugins***" menu to see if the menu item we created is there. And indeed, we can see that the "APT32 Graph Deobfuscator" item now appears in the menu.

Fig 6: *The menu item we created was successfully added*

We can now choose some function which we suspect for having these junk blocks, and try to test our Plugin. In this example, We chose the function `fcn.00acc7e0`. Going to a function in Cutter can be done either by selecting it from the left menu, or simply pressing “g” and typing its name or address in the navigation bar.

Make sure you are in the graph view. Feel free to wander around and try to spot the junk blocks. We highlighted them in the image below which shows the Graph Overview (mini-graph) window.

Fig 7: *Junk block highlighted in* `fcn.00acc7e0`

Since we have a candidate suspicious function, we can trigger our plugin and see if it successfully removes them. To do this, click on "***Windows -> Plugins -> APT32 Graph Deobfuscator***". After a second, we can see that our plugin successfully removed the junk blocks.

Fig 8: *The same function after removing the junk blocks*

In the following images, you can see more pairs of function graphs before and after the removal of junk blocks.

Fig 9: *Before and After of* `fcn.00aa07b0`

Fig 10: *Before and After of* `fcn.00a8a1a0`

Final Words

Ocean Lotus' obfuscation techniques are in no way the most complicated or hard to beat. In this article we went through understanding the problem, drafting a solution and finally implementing it using the python scripting capabilities of Cutter and Radare2. The full script can be found in our Github repository, and also attached to the bottom of this article.

If you are interested in reading more about Ocean Lotus, we recommend this high-quality article published by ESET's Romain Dumont. It contains a thorough analysis of Ocean Lotus' tools, as well as some exposition of the obfuscation techniques involved.

Appendix

Sample SHA-256 values

- Be6d5973452248cb18949711645990b6a56e7442dc30cc48a607a2afe7d8ec66
- 8d74d544396b57e6faa4f8fdf96a1a5e30b196d56c15f7cf05767a406708a6b2

APT32 Graph Deobfuscator – Full Code

```
""" A plugin for Cutter and Radare2 to deobfuscate APT32 flow graphs
This is a python plugin for Cutter that is compatible as an r2pipe script for
radare2 as well. The plugin will help reverse engineers to deobfuscate and remove
junk blocks from APT32 (Ocean Lotus) samples.
"""

__author__  = "Itay Cohen, aka @megabeets_"
__company__ = "Check Point Software Technologies Ltd"

# Check if we're running from cutter
try:
    import cutter
    from PySide2.QtWidgets import QAction
    pipe = cutter
    cutter_available = True
# If no, assume running from radare2
except:
    import r2pipe
    pipe = r2pipe.open()
    cutter_available = False


class GraphDeobfuscator:
    # A list of pairs of opposite conditional jumps
    jmp_pairs = [
        ['jno', 'jo'],
        ['jnp', 'jp'],
        ['jb',  'jnb'],
        ['jl',  'jnl'],
        ['je',  'jne'],
        ['jns', 'js'],
        ['jnz', 'jz'],
        ['jc',  'jnc'],
        ['ja', 'jbe'],
        ['jae', 'jb'],
        ['je',  'jnz'],
        ['jg',  'jle'],
```

```
        ['jge', 'jl'],
        ['jpe', 'jpo'],
        ['jne', 'jz']]

    def __init__(self, pipe, verbose=False):
        """an initialization function for the class

        Arguments:
            pipe {r2pipe} -- an instance of r2pipe or Cutter's wrapper

        Keyword Arguments:
            verbose {bool} -- if True will print logs to the screen (default: {False})
        """

        self.pipe = pipe

        self.verbose = verbose

    def is_successive_fail(self, block_A, block_B):
        """Check if the end address of block_A is the start of block_B

        Arguments:
            block_A {block_context} -- A JSON object to represent the first block
            block_B {block_context} -- A JSON object to represent the second block

        Returns:
            bool -- True if block_B comes immediately after block_A, False otherwise
        """

        return ((block_A["addr"] + block_A["size"]) == block_B["addr"])

    def is_opposite_conditional(self, cond_A, cond_B):
        """Check if two operands are opposite conditional jump operands

        Arguments:
            cond_A {string} -- the conditional jump operand of the first block
            cond_B {string} -- the conditional jump operand of the second block

        Returns:
            bool -- True if the operands are opposite, False otherwise
        """

        sorted_pair = sorted([cond_A, cond_B])
        for pair in self.jmp_pairs:
            if sorted_pair == pair:
                return True
        return False
```

```
 85.
 86.     def contains_meaningful_instructions (self, block):
 87.         '''Check if a block contains meaningful instructions (references, calls, strings,...)
 88.
 89.         Arguments:
 90.             block {block_context} -- A JSON object which represents a block
 91.
 92.         Returns:
 93.             bool -- True if the block contains meaningful instructions, False otherwise
 94.         '''
 95.
 96.         # Get summary of block - strings, calls, references
 97.         summary = self.pipe.cmd("pdsb @ {addr}".format(addr=block["addr"]))
 98.         return summary != ""
 99.
100.     def get_block_end(self, block):
101.         """Get the address of the last instruction in a given block
102.
103.         Arguments:
104.             block {block_context} -- A JSON object which represents a block
105.
106.         Returns:
107.             The address of the last instruction in the block
108.         """
109.
110.         # save current seek
111.         self.pipe.cmd("s {addr}".format(addr=block['addr']))
112.         # This will return the address of a block's last instruction
113.         block_end = self.pipe.cmd("?v $ @B:-1")
114.         return block_end
115.
116.     def get_last_mnem_of_block(self, block):
117.         """Get the mnemonic of the last instruction in a block
118.
119.         Arguments:
120.             block {block_context} -- A JSON object which represents a block
121.
122.         Returns:
123.             string -- the mnemonic of the last instruction in the given block
124.         """
125.
126.         inst_info = self.pipe.cmdj("aoj @ {addr}".format(addr=self.get_block_end(block)))[0]
127.         return inst_info["mnemonic"]
128.
129.     def get_jump(self, block):
130.         """Get the address to which a block jumps
131.
```

```
132.         Arguments:
133.             block {block_context} -- A JSON object which represents a block
134.
135.         Returns:
136.             addr -- the address to which the block jumps to. If such address doesn't exist, re
137.         """
138.
139.         return block["jump"] if "jump" in block else None
140.
141.     def get_fail_addr(self, block):
142.         """Get the address to which a block fails
143.
144.         Arguments:
145.             block {block_context} -- A JSON object which represents a block
146.
147.         Returns:
148.             addr -- the address to which the block fail-branches to. If such address doesn't e
149.         """
150.         return block["fail"] if "fail" in block else None
151.
152.     def get_block(self, addr):
153.         """Get the block context in a given address
154.
155.         Arguments:
156.             addr {addr} -- An address in a block
157.
158.         Returns:
159.             block_context -- the block to which the address belongs
160.         """
161.
162.         block = self.pipe.cmdj("abj. @ {offset}".format(offset=addr))
163.         return block[0] if block else None
164.
165.     def get_fail_block(self, block):
166.         """Return the block to which a block branches if the condition is fails
167.
168.         Arguments:
169.             block {block_context} -- A JSON representation of a block
170.
171.         Returns:
172.             block_context -- The block to which the branch fails. If not exists, returns None
173.         """
174.         # Get the address of the "fail" branch
175.         fail_addr = self.get_fail_addr(block)
176.         if not fail_addr:
177.             return None
178.         # Get a block context of the fail address
```

```
179.         fail_block = self.get_block(fail_addr)
180.         return fail_block if fail_block else None
181.
182.     def reanalize_function(self):
183.         """Re-Analyze a function at a given address
184.
185.         Arguments:
186.             addr {addr} -- an address of a function to be re-analyze
187.         """
188.         # Seek to the function's start
189.         self.pipe.cmd("s $F")
190.         # Undefine the function in this address
191.         self.pipe.cmd("af- $")
192.
193.         # Define and analyze a function in this address
194.         self.pipe.cmd("afr @ $")
195.
196.     def overwrite_instruction(self, addr):
197.         """Overwrite a conditional jump to an address, with a JMP to it
198.
199.         Arguments:
200.             addr {addr} -- address of an instruction to be overwritten
201.         """
202.
203.         jump_destination = self.get_jump(self.pipe.cmdj("aoj @ {addr}".format(addr=addr))[0])
204.         if (jump_destination):
205.             self.pipe.cmd("wai jmp 0x{dest:x} @ {addr}".format(dest=jump_destination, addr=add
206.
207.     def get_current_function(self):
208.         """Return the start address of the current function
209.
210.         Return Value:
211.             The address of the current function. None if no function found.
212.         """
213.         function_start = int(self.pipe.cmd("?vi $FB"))
214.         return function_start if function_start != 0 else None
215.
216.     def clean_junk_blocks(self):
217.         """Search a given function for junk blocks, remove them and fix the flow.
218.         """
219.
220.         # Get all the basic blocks of the function
221.         blocks = self.pipe.cmdj("afbj @ $F")
222.         if not blocks:
223.             print("[X] No blocks found. Is it a function?")
224.             return
225.         # Have we modified any instruction in the function?
```

```
226.         # If so, a reanalyze of the function is required
227.         modified = False
228.
229.         # Iterate over all the basic blocks of the function
230.         for block in blocks:
231.             fail_block = self.get_fail_block(block)
232.             # Make validation checks
233.             if not fail_block or \
234.             not self.is_successive_fail(block, fail_block) or \
235.             self.contains_meaningful_instructions(fail_block) or \
236.             not self.is_opposite_conditional(self.get_last_mnem_of_block(block), self.get_last
237.                 continue
238.             if self.verbose:
239.                 print ("Potential junk: 0x{junk_block:x} (0x{fix_block:x})".format(junk_block=
240.             self.overwrite_instruction(self.get_block_end(block))
241.             modified = True
242.         if modified:
243.             self.reanalize_function()
244.
245.     def clean_graph(self):
246.         """the initial function of the class. Responsible to enable cache and start the cleani
247.         """
248.
249.         # Enable cache writing mode. changes will only take place in the session and
250.         # will not override the binary
251.         self.pipe.cmd("e io.cache=true")
252.         self.clean_junk_blocks()
253.
254.
255. if cutter_available:
256.     # This part will be executed only if Cutter is available. This will
257.     # create the cutter plugin and UI objects for the plugin
258.     class GraphDeobfuscatorCutter(cutter.CutterPlugin):
259.         name = "APT32 Graph Deobfuscator"
260.         description = "Graph Deobfuscator for APT32 Samples"
261.         version = "1.0"
262.         author = "Itay Cohen (@Megabeets_)"
263.
264.         def setupPlugin(self):
265.             pass
266.
267.         def setupInterface(self, main):
268.             # Create a new action (menu item)
269.             action = QAction("APT32 Graph Deobfuscator", main)
270.             action.setCheckable(False)
271.             # Connect the action to a function - cleaner.
272.             # A click on this action will trigger the function
```

```
            action.triggered.connect(self.cleaner)

            # Add the action to the "Windows -> Plugins" menu
            pluginsMenu = main.getMenuByType(main.MenuType.Plugins)
            pluginsMenu.addAction(action)

        def cleaner(self):
            graph_deobfuscator = GraphDeobfuscator(pipe)
            graph_deobfuscator.clean_graph()
            cutter.refresh()


    def create_cutter_plugin():
        return GraphDeobfuscatorCutter()


if __name__ == "__main__":
    graph_deobfuscator = GraphDeobfuscator(pipe)
    graph_deobfuscator.clean_graph()

```

Source: https://research.checkpoint.com/deobfuscating-apt32-flow-graphs-with-cutter-and-radare2/