Abstract

For many years, one of the great hazards to computers is malware which can cause a dramatic impact on their victim. Meanwhile, the number of new malware has increased exponentially each year, and most of them are obfuscated by packers for protecting malware from different anti-virus systems and many algorithms.

The Original Entry Point (OEP) of packed code is the place indicating the beginning point of the original payload during the execution of packed code. When OEP is detected, we can observe and investigate the original functionality. However, finding OEP automatically is still a challenging task because of the diversity of packers. In some previous works, OEP can be found in the specific region with some patterns (e.g., jump to the reserved region), but finding these patterns is still difficult for many packers. Therefore, a method to automatically detect OEP is essential and meaningful.

Based on our observation, most of packers will encrypt the original payload and combine it with an unpacking stub, which decrypts the packed payload during the execution of the packed code. After the last instruction in the unpacking stub is executed, the control flow will jump to the OEP. Hence, the unpacking stub can be regarded as a signature for each packer and we expect some pattern in each stub. To find such patterns, we observe CFG of various packed codes, which is generated by BE-PUM (Binary Emulation for PUshdown Model), a dynamic symbolic execution tool of Intel x86 binaries. We store the template of each unpacking stub by applying BE-PUM on example-packed codes. Since we know the original code, we can compare the original and the restored code by BE-PUM, and obtain the CFG of the unpacking stub as the difference.

This paper proposes a method for packer identification and OEP (Original Entry Point) detection based on the graph similarity on control flow graphs of packed codes. Packed code consists of an unpacking stub and a packed payload, which is recovered to the original after the unpacking stub execution. We start with the hypothesis that the CFG of the unpacking stub characterizes a packer. The CFGs of packed code are generated by a DSE (Dynamic Symbolic Execution) tool BE-PUM on x86-32/Windows, and when their original payloads and used packers are known, we can identify the unpacking stub in a packed code. First, for 771 samples packed by 12 packers from given payloads, we classify them by a clustering algorithm DBSCAN to confirm the hypothesis. We observe that when the allowance *eps* is enough small (e.g., $eps = 0.02$ in DBSCAN), (1) each class does not cross different packers (whereas some packer has multiple classes of CFGs of the unpacking stubs, e.g., WINUPACK has 2), and (2) the end instruction sequence (the prefix of the exit of an unpacking stub with the specified length) is the same in each class. Hence, we define the *template* of the class as the pair of the average of Weisfeiler-Lehman histogram vectors and the end sequence. Each template is computed packer-wise (i.e., clustering packed codes by the same packer) for the ease to cover a new packer. Next, for unknown packed code, BE-PUM incrementally generates the CFG. When the end sequence matches with the tail of a CFG fragment, we check the similarity

between its Weisfeiler-Lehman histogram vector and that in templates. Among them, the CFG fragment with the highest cosine similarity is regarded as the unpacking stub, which also detects the used packer and the OEP as the jump destination from the exit. Our experiment focuses on 12 packers, UPX, ASPACK, FSG, YODA'S Crypter, MEW, PACKMAN, PECOMPACT, PETITE, (WIN)UPACK, JDPACK, MPRESS, TELOCK, and the OEP of 688 among 700 non-malware packed samples (of which the original payload is also known) is correctly detected. Further, we apply the method to 1239 malware samples. Among them, 1089 samples are detected packed and 150 samples are packed by the 11 packers (except for TELOCK). We conclude that our method is highly effective as long as we have access to an executable of a target packer to compute its templates.

Keywords— Original Entry Point, x86 malware, packer, clustering algorithm, graph kernel, graph theory.