A Generic Approach to Deobfuscation*
Babak Yadegari, Brian Johannesmeyer, Benjamin Whitely, Saumya Debray

*In proceedings of IEEE S&P’15

Motivations and Contributions
- Malicious software are usually obfuscated to avoid detection and resist analysis.
- Such obfuscations have to be penetrated or removed (“deobfuscated”) in order to understand the internal logic of the code and devise countermeasures.
- Much of the current approaches rely on obfuscation-specific techniques so are not effective against unseen and new obfuscations.
- We are presenting a semantic-based approach that makes very few assumptions about the underlying obfuscation technique(s) and is so is effective against future obfuscations.

Idea
- Model program as a mapping from its inputs to outputs, identify the computation that affects the observable behavior and apply semantic-preserving transformations to simplify this mapping.
- Input/Outputs are identified through the program System calls, or the information set by the operating system.

Deobfuscation Overview
- Malicious code often involves self-modifying and/or dynamically unpacked code, we use dynamic analysis.
- Record Execution Trace
- Forward/Backward Taint Analysis
- Identify Control-Dependancies
- Semantic-Preserving Simplifications
- Construct Final CFG

Our Approach
- Due to the obfuscation, standard taint propagation techniques, fail to produce precise results and usually overtaint. We propagate taint at bit-level granularity, use distinct taint sources, and use instruction semantics as propagation policy to avoid overtainting.
- Original control dependencies might be obscured by the obfuscation, e.g., due to CFG Flattening or Virtualization: we need to consider the set of control transfers that are affected by implicit data flows (we call this set implicit control flows).
- The simplification step includes a series of semantic-preserving transformations which are a generalized form of compiler optimizations, e.g.:
  - Constant folding
  - Dead code elimination
  - Copy propagation
- Reuse of the same piece of code causes spurious execution paths in the final CFG, e.g. the same piece of code is used for every add instructions in a virtualized obfuscated code. We try to minimize spurious paths by replicating basic blocks in the final CFG.

Results
- Obfuscation techniques used to evaluate: single- and multi-level virtualization and ROP.
- Test-cases include both synthetic and malicious programs, they were obfuscated using commercial obfuscation tools, e.g. Themida, VMProtect etc. for virtualization ROPed manually.

Some sample results:
- Binary-search program obfuscated using Themida
- Fibonacci program written in ROP

Acknowledgments
This research was supported in part by the National Science Foundation (NSF) under grants CNS-1115829, CNS-1145913, IIS-1318343, and CNS-1318955, and the US Department of Defense under grant no. FA2386-14-1-3016; and AFOSR grant no. FA9550-11-1-0191