Dr. Tim Blazytko
M. Sc. Philipp Koppe

Lecture Software Protection
Summer Term 2025
Ruhr-Universität Bochum

Task Sheet 2

April 30, 2024
Submission due to June 8, 2025, 11:59 PM via Moodle.
For question, feel free to ask via Discord or Moodle.

Tasks

Course assessment is based on 4-5 task sheets which will be completed within the semester. The final grade consists of the weighted average grade of all task sheets. We will report every student to the examination office who handed in at least two tasks. In other words, we allow students to drop out after handing in a single task. Task sheets have to be worked on individually; group work is not possible.

Optional tasks are voluntary and independent of the course assessment; in other words, they do not influence the final grade. Instead, they are designed for students who want to dive deeper into individual topics. These submissions will be ranked with subjective score points. The student who has the highest score at the end of the semester wins a Binary Ninja student license.

The second task sheet focuses on analyzing and breaking real-world (inspired) obfuscation schemes (opaque predicates and VMs), but also covers some additional use cases of SMT solvers. It also includes the analysis of a VM found in Microsoft products. The optional exercise dives deeper into this analysis.

For the practical implementation tasks, it is highly recommended to use the Miasm reverse engineering framework. While an older, outdated stable version can be installed via PIP, it is recommended to choose (one of the) latest GitHub commits.

Please submit your solutions as either text files, if several files have to be submitted, as zip files. Please do not hand in `.doc` or similar formats. If you have to include images or need some special document layouting, export the document as pdf file.

Task 1: Opaque Predicates II (4 Points)

The function `0x400546` (in `opaque_predicates`) protects some input-dependent calculations within a large number of opaque predicates. You can assume that all opaque predicates are local within a single basic block.

(a) Write a Python script which identifies all opaque predicates in the function. Optionally, you can also use the script to patch all opaque predicates in the binary. (1 Point)

(b) Reconstruct the function's high-level code (in C-like pseudocode) with all opaque predicates removed. If you patched the binary, an annotated decompiler output is also sufficient. (2 Points)

(c) Use an SMT solver to find at least one pair of inputs for which the binary prints "Correct!". You can integrate it in your opaque predicate detection script or write a new script. (1 Point)

Comment your code. Submit your Python script(s), the reconstructed code and, if existing, your patched binary. Also include the output of an example run (in a text file).

Task 2: VM Disassembler (16 Points)

Your goal is to write a disassembler for the VM in the function `0x11a0` (in `vm.bin`) and reconstruct the protected high-level code. To achieve this, proceed as follows:

(a) Identify VM architecture (register/stack-based) and components (virtual instruction pointer, bytecode location, handler locations, ...). Describe your findings. (1 Point)

(b) Identify the individual handler semantics and their opcodes. (3 Points)

(c) Write a Python script which, based on symbolic execution, automatically follows the VM execution flow and uses the bytecode as guidance. Hardcode symbolic registers and memory locations where necessary. (3 Points)

(d) Add callbacks for individual VM handlers and dump relevant handler information. These will be your output of your VM disassembler (5 Points)

(e) Manually reconstruct the protected high-level code in C-like pseudocode (based on the output from your VM disassembler). The function inputs 2, 9 and 12 are sufficient to obtain 100% coverage of the original code. (4 Points)

Comment your analysis results, design decisions and code. Submit your notes, VM disassembler and reconstructed high-level code. Also include the output of an example run (in a text file).

Task 3: Real-World VM Analysis I (4 Points)

`ci.dll` is a kernel-mode library which implements security-critical mechanisms for Microsoft Windows. Like many other Microsoft binaries (e.g., DRM libraries or even the kernel itself), it uses a custom virtual machine to protect code against analysis. Your goal is to locate the VM and analyze some of its components.

(a) Locate the virtual machine in `ci.dll`. In how many functions is it used? (1 Point)

(b) Locate the virtual instruction pointer and the handler table. How many handlers does the VM have? (1 Point)

(c) The VM design differs in various aspects from the ones we discussed so far. Name and describe at least two. (2 Points)

Describe your findings and analysis notes in a text file. Refer to concrete code locations and describe how you found them.

Optional Task: Real-World VM Analysis II

In this task, you will dive deeper into Microsoft's in-house virtual machine. This can be realized in various ways:

(a) Perform a deeper analysis of the virtual machine in `ci.dll`. How does it work internally? What is it used for?

(b) Write a VM disassembler (based on symbolic execution) for the VM. While comparable to the VM disassembler for `vm.bin`, you have to take care of API functions, function calls etc. In practice, it is one of the hardest challenges to engineer analysis tooling for real-world binaries.

(c) Look out for VM instances in other Microsoft products. Starting points may be printer drivers, DRM in Office products and others.

No matter which tasks you choose, feel free to reach out for ideas, implementation guidance etc. Submit a text file with your analysis results as well as potential codes or analysis binaries.