Intrusion Detection Problem Set

Description

The goal of this problem set is to develop a host-based intrusion detection system for a vulnerable program to defend it against exploitation.

To complete the problem set, you will need to ssh to your container at `$user@amplifier.ccs.neu.edu:$port`, where `$user` is your gitlab username and `$port` is your assigned ssh port (https://seclab-devel.ccs.neu.edu/snippets/6). Authentication is performed using any of your uploaded ssh public keys in gitlab.

You will also need to clone the problem set repository located at `git@seclab-devel.ccs.neu.edu:softvulnsec/prset03.git`.

Important Information	
Available	Fri 27 Feb 20:00 EST
Submission Deadline	Fri 06 Mar 18:00 EST
Gitlab URL	https://seclab-devel.ccs.neu.edu/softvulnsec/prset03 (https://seclab-devel.ccs.neu.edu/softvulnsec/prset03)

Syscall Sequence Model

For this problem, you will need to build a model that encodes the possible sequences of system calls that can be issued by the vulnerable program `prset03`. To collect these sequences, you will use dynamic tracing using `strace`, which outputs the system calls issued during an execution of a program. For instance, consider the following trace:

```
$ strace -f -o prset03.strace ./prset03
$ cat prset03.strace
8008  execve("./prset03", ["./prset03"], [/* 15 vars */]) = 0
8008  brk(0)                                = 0x1131000
8008  access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
8008  mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f1792f97000
8008  access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
8008  open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[...]
```

Here, `strace` indicates that on this run of the program, the initial sequence of system calls is `execve` → `brk` → `access` → `mmap` → `access` → `open`.

The particular model that you will build is a digram model, which records pairs of system calls that appear adjacent to one another in a collected trace. A digram model is simply an *n*-gram model where *n*=2 which is constructed by sliding a window of size *n* over a sequence.

From the initial sequence above, the resulting digram model would be:

```
(execve, brk)
(brk, access)
(access, mmap)
(mmap, access)
(access, open)
```

Write a program that parses the output of `strace` on `prset03` to produce a digram model. Your program should output the model as JSON with the following format:

```
{
    "model": [
        {
            "syscall_1": "<first syscall>",
            "syscall_2": "<second syscall>"
        },
        // ...
    ]
}
```

As an example, the JSON representation of the model of the initial sequence above would be:

```
{
    "model": [
        {
            "syscall_1": "execve",
            "syscall_2": "brk"
        },
        {
            "syscall_1": "brk",
            "syscall_2": "access"
        },
        {
            "syscall_1": "access",
            "syscall_2": "mmap"
        },
        {
            "syscall_1": "mmap",
            "syscall_2": "access"
        },
        {
            "syscall_1": "access",
            "syscall_2": "open"
        }
    ]
}
```

Important

Make sure that your `strace` parser handles the output of `strace -o $output_path -f`, where `-f` instructs `strace` to follow forks and `-o $path` redirects the trace to a file.

Model Coverage

The model produced from a single execution trace from `prset03` is not necessarily *sound* – i.e., it only covers one possible path through the program, and therefore might not contain legal syscall pairs that could be issued on other paths that weren't covered in the first trace.

Examine the source code to `prset03` to identify program inputs to cover all possible paths through the application. Provide those inputs to the program to collect a set of syscall traces using `strace`. From this set of traces, build a model that is sound with respect to path coverage of the code comprising `prset03`.

Note

This notion of soundness does not account for uncovered paths in library code. We'll ignore that issue for this problem set.

Mimicry Attacks

Examine the digram model built from `prset03` in the previous problem. Is it possible to modify your attack from problem set 1 to perform a mimicry attack that evades detection by your model? Why or why not?

Augmented Syscall Sequence Model

One approach to making the digram syscall model "tighter" is to add syscall arguments to the model. In this problem, you will augment the model you created in previous problems with partial argument whitelists.

Let us consider the initial syscall sequence from above. In addition to recording the syscalls themselves, we can also attempt to model an upper bound on the allowed arguments. For example, by converting any concrete string to a prefix pattern like so:

```
(execve ./prset03, brk)
(brk, access /etc/)
(access /etc/, mmap)
(mmap, access /etc/)
(access /etc/, open /etc/)
```

Update your model builder program to produce an augmented model from `strace` output in the style above as a JSON object with the following format:

```
{
    "model": [
        {
            "syscall_1": "<first syscall>",
            "syscall_2": "<second syscall>",
            "syscall_1_arg": "<first syscall argument pattern|null>",
            "syscall_2_arg": "<second syscall argument pattern|null>"
        },
        // ...
    ]
}
```

Compute argument prefixes *when merging digram entries* by taking the longest common prefix between two strings. For instance, given:

```
(brk, access /etc/ld.so.nohwcap)
(brk, access /etc/ld.so.cache)
```

the merge would be

```
(brk, access /etc/ld.so.)
```

As an example, the JSON representation of the augmented initial sequence above after merging might be:

```
{
    "model": [
        {
            "syscall_1": "execve",
            "syscall_2": "brk",
            "syscall_1_arg": "./prset03",
            "syscall_2_arg": null
        },
        {
            "syscall_1": "brk",
            "syscall_2": "access",
            "syscall_1_arg": null,
            "syscall_2_arg": "/etc/"
        },
        {
            "syscall_1": "access",
            "syscall_2": "mmap",
            "syscall_1_arg": "/etc/",
            "syscall_2_arg": null
        },
        {
            "syscall_1": "mmap",
            "syscall_2": "access",
            "syscall_1_arg": null,
            "syscall_2_arg": "/etc/"
        },
        {
            "syscall_1": "access",
            "syscall_2": "open",
            "syscall_1_arg": "/etc/",
            "syscall_2_arg": "/etc/"
        }
    ]
}
```

Mimicry Attacks Redux

Examine the augmented digram model that you built in the previous problem. Is it possible to modify your attack from problem set 1 to perform a mimicry attack that evades detection by the *augmented* model? Why or why not?

Attack Detection

Write a tool that evaluates a system call trace produced by `strace` against a model. Your tool should output a JSON object with the following format:

```
{
    "malicious": <true|false>,
    "syscall": <malicious syscall name|null>,
    "syscall_arg": <malicious syscall argument|null>
}
```

where `malicious` is a boolean value indicating whether the trace contains evidence of malice, `syscall` is either the name of the (first) malicious syscall as a string or `null` if the trace was benign, and `syscall_arg` is an associated malicious argument value if the trace was malicious *and* a malicious argument was observed, otherwise `null` . For instance, a benign trace might result in the following output:

```
{
    "malicious": false,
    "syscall": null,
    "syscall_arg": null
}
```

whereas a malicious trace might result in:

```
{
    "malicious": true,
    "syscall": "open",
    "syscall_arg": "/etc/shadow"
}
```

Answer Submission

Fork the repository for this problem set in gitlab – your copy should have the URL `git@seclab-devel.ccs.neu.edu:$user/prset03.git`. In your forked repository, commit a JSON object to `solution.json` with the following format:

```
{
    "model": [
        {
            "syscall_1": "<first syscall>",
            "syscall_2": "<second syscall>"
        },
        // ...
    ],
    "mimicry_possible": <true|false>,
    "model_augmented": [
        {
            "syscall_1": "<first syscall>",
            "syscall_2": "<second syscall>",
            "syscall_1_arg": "<first syscall argument pattern|null>",
            "syscall_2_arg": "<second syscall argument pattern|null>"
        },
        // ...
    ],
    "mimicry_augmented_possible": <true|false>
}
```

where

- `model` is the combined digram model constructed from traces collected over all possible paths through `prset03`;
- `mimicry_possible` is a `true` or `false` answer to the question of whether a mimicry attack is possible against `model`;
- `model_augmented` is the digram model augmented with syscall argument patterns; and
- `mimicry_augmented_possible` is a `true` or `false` answer to the question of whether a mimicry attack is possible against `model_augmented`.

You are responsible for submitting valid JSON at the correct path. Use a validator if you're unsure about this, and double-check that your JSON follows the format above exactly.

In addition, commit the source code to your model builder and enforcement tool to `model/` in your repository.

Commit an executable script called `checker` to the root of your repository that accepts a path to `solution.json` as the first argument and a path to a system call trace produced by `strace`. This script should invoke your model enforcer and produce the JSON enforcement decision described above. For example, your script should operate like the following:

```
$ ./checker /path/to/solution.json /path/to/strace_output
{
    "malicious": false,
    "syscall": null,
    "syscall_arg": null
}
```

Finally, commit a `README.md` that describes how your model builder and enforcer works, and justification for your two answers to the mimicry attack questions.

Push all commits to gitlab.

Your solution will be scored in terms of the following:

- Correctness of the models you generate
  - Whether they fully cover the set of possible syscall pairs, and
  - Whether they accurately model system call arguments
- Whether your answers to the mimicry attack questions are correct and well-justified
- Whether your model enforcer correctly identifies attacks

Extra Credit

Another approach to improving the fit of the model is to build a finite state automaton (FSA) instead of a digram model. Following the control flow graph for `prset03`, build an FSA that encodes an over-approximation of the possible benign sequences of system calls *and* argument patterns. Use the following JSON format as part of `solution.json` to receive credit:

```
{
    // original fields omitted
    "model_fsa": [
        {
            // synthetic start node
            "id": 0,
            "syscall": null,
            "arg": null,
            "successors": [
                <list of node IDs>
            ]
        },
        {
            "id": 1,
            "syscall": "execve",
            "arg": "./prset03",
            "successors": [
                2
            ]
        },
        {
            "id": 2,
            "syscall": "brk",
            "arg": null,
            "successors": [
                3
            ]
        },
        // ...
    ]
}
```

Updated Fri 07 Aug 2015 10:38 EDT
Revision master/52d826a