Reverse Engineering for the Detection of Undesirable Functionality in Commercial Software

Problem Description

Software consumers generally employ software with only a basic idea of the functions that it performs. A certain amount of trust is given in assuming that the software will perform as advertised, doing no less and no more than expected. Software packages are, by nature, very complex, and it is difficult to discern all of the actions that are performed when the software is executed. It is very possible that commercially available software might contain undesirable functionality (UF) that is not easily detectable in normal use. This UF may be either intentional or unintentional on the part of the software producer. In some instances the UF may be of a nature that would compromise the security of the network or the information being handled. Intentional UF may be designed to collect information on the use of the product, subvert security, divert information to unauthorized parties, or collect keys, passwords or other data for retrieval by unauthorized parties.

Solution

UF can be detected by applying a reverse engineering process to the code and deriving the total set of functions that that code will perform. Any software can be reverse-engineered to determine exactly what functionality is present. Since resources are limited, it is desirable to have a carefully controlled process that targets the effort toward high-risk sections of the software and exploits opportunities for automation. The process described here identifies the most suspect portions of code for study, and then utilizes formal methods to abstract the program function and presents those results for analysis.

Undesirable Functionality Characteristics

Undesirable functionality is specific behavior that permits non-secure access, exports sensitive information, damages computer or network operations, or otherwise compromises system security. Intentional UF is a result of code that has been deliberately inserted or designed into a system to breach system security. There is a potentially infinite set of unique instances of UF; therefore a method that requires a search for specific functions or their variants is too limited to give confidence. A more general approach can be employed that exploits the basic characteristics of information processing. By examining the functional behavior of a software system, it is possible to determine what data is accepted as input to a system and where it comes from, what processing or data manipulation is performed, and what data is emitted and to where it goes. UF then, falls into three categories:

- It emits data from the system to an undesired recipient
- It accepts data or commands from an undesired source
- It manipulates data in some way that is misrepresentative or destructive

The data emitter category generally refers to the dissemination of data to an unauthorized party. Data can be "emitted" from a program in a number of ways. To determine if a particular data emission is undesirable, we must examine the type of information being emitted, the destination of the emission, and the circumstances under which the emission occurs.

The reference to data acceptors primarily refers to the acceptance of commands in this context, though a data acceptor could be used to misrepresent data from an undesirable source as authentic (spoofing). UF of this type will become evident through reverse engineering as input sources are identified, and the process for authenticating input sources is examined.

A data manipulator is functionality that performs modification of data in the system in some way to introduce error, destroy data, or otherwise weaken the integrity of the system (sabotage). All data manipulation that occurs in a software system will be visible to reverse engineering as the program function is extracted. Any manipulation that is outside of the advertised function of the software can be identified and thoroughly examined.

In general, these types of behavior are also present in desirable functionality. It is only the context of the behavior that determines its negative impact. In order to understand the behavior in context, it is necessary to examine the total functionality present in the software.

The UF Identification Process

Work performed by Wisdom Software, Ltd. has resulted in the definition of the fundamental processes needed for the efficient identification of UF. UF is evolving and will continue to evolve. These processes are intended to address the general problem of UF detection by revealing the fundamental functionality of the code. Using these processes, source code can be broken down and analyzed to reveal the functionality of the code in a format usable by a human analyst. Functionality is abstracted using a formal reverse engineering process. The process begins by defining the functionality of the lowest level segments of the software. Functional composition is then used to combine lower level segment definitions into higher-level definitions, identifying the higher-level functionality. The process produces a clear definition of response to the possible combinations of stimuli both in the form of a state box definition, and as a "legal stimulus sequence" table.

Example Code

```
    {
        if (waitpid(p->pid, (int *) 0, 1 ) > 0) {
            p->kill_how = kill_never;
        }
    }
```

Corresponding State box

Stimulus	Current Condition	State Update	Response
Invoke		null	waitpid(p->pid, (int*) 0, 1)
waitpid(p->pid, (int*) 0, 1) > 0		p->kill_how = kill_never	terminate
waitpid(p->pid, (int*) 0, 1) ≤ 0		null	terminate

The function abstraction process makes use of the fact that every program is simply a rule for a function. By identifying the functional rule for small pieces of software, it is possible to derive the rule for larger composite items of software. A complete function abstraction process has been defined and documented that addresses all software constructs, and by picking a set number of iterations can deal with dynamic iterative constructs such as while loops. The process is generally applicable to any software, but has been specifically implemented for "C" code in our initial research research. The process is tedious, but amenable to automated support.

Once the program function has been abstracted and expressed in the form of a state box, it is possible to generate a list of possible input sequences with their associated responses and state updates. This makes the entire activity of the program function visible. A large amount of data is generated in this step, and the efficient analysis of the resulting data is a subject of current research. Analysis of the functional data developed by the reverse engineering processes is expected to always require the intervention of a human analyst.

With the completed function abstraction, the analyst has powerful new insight into the inner workings of a software system:

1. The complete program function is fully documented in a hierarchy of state box tables
2. The combination of program actions, the effective behavior of the software, can be easily understood from the legal sequence tables. These tables can be produced for individual segments, collections of composed segments, or for the entire software system.
3. Dead code can be identified automatically. The questions of why dead code is left in a program may raise suspicion of its purpose.

Segment Prioritization

Even though much of the function abstraction and UF search process is currently being automated, final analysis will still require a large amount of data to be examined by a human analyst. The analyst may have specific types of UF in mind or may be conducting a more general search. It is very important that he or she be able to review small portions, segments, of the code in a prioritized order. This is the most cost effective use of the analyst's time, and will provide the analyst with the best possible insight for detecting UF.

A variable weighting scheme has been initially defined to establish the priority for the segments. This weighting scheme is based on the fundamental characteristics of the code and the state box definition of the function. The analyst may define the initial set of weights or select from predetermined sets. Once the abstraction and analysis process has begun, the analyst can go back and re-prioritize the remaining segments based on knowledge gained from the analysis. The impact of the weighting scheme on the probability of detection for specific UF still requires further research, but this approach should provide the most cost effective way of examining the critical parts of large software systems.

Conclusions

The functional behavior of software can be revealed and analyzed to determine if it is correct and safe. Today, the reverse engineering process is prohibitively expensive and produces large quantities of data that is difficult to interpret. The research being conducted by Wisdom Software, Ltd. will provide the theory and the tooling to conduct the analysis efficiently and develop a meaningful measure of confidence that the software is safe to use.