Summary of *Protecting Software Code By Guards*

Anna Segurson

March 7, 2002

1 Introduction

Although most people think a cracker is unlevened bread sprinkled with salt, *crackers* are actually a serious threat to the software industry. These *crackers* obtain software that have mechanisms installed to prevent illeagal distribution and produce *crackz*, modified versions of the software with the preventive mechanisms disabled. These *crakz* can then be distributed without any profit to the original distributor of the software.

The authors believe that *crackers* have an easy time "cracking" most binaries (executables) because software protection in place has a single point of failure. For example, the attacker may find a set of instructions like

```
if (key is not valid) die.
```

Once this is identified all the attacker would have to do remove that single `if`-statement, or throw an unconditional jump in directly before the `if`-statement that skips right over it. The authors define monolithic protection schemes to be proctection schemes implemented in a single code module, but only loosely attached to the program. These monolithic protection schemes can also be easily disabled by identify the single code block and removing all calls to it in the rest of the program.

Any scheme used to protect software can eventually be broken. The goal is to make it as hard as possible for the *crackers* to produce *crakz*. The software company wants to sell as many copies of their product as possible before the illegal copies are produced. To achieve this the authors identified four properties that a *sufficiently secure* protection scheme should entail:

- **Resilience**: hard to disable
- **Self-defense** detects tampering and repairs itself
- **Configurability** amount of protection is customizable
- **White-box Security** since the algorithm will be known, it should depend on some random seed to install so the protection code isn't easily reproducable (thus easily removable).

All of these ideas are embodied in a network of guards. Guards are functional units of code that protect critical units of code (which can include other guards). The guards are distributed through a program elimiating the single point of failure, they can fix and/or check code, they are automaticly inserted in a binary with the number of guards being user-configurable, and the topology of the guard network is not the same for all inputs.

2 Similar Work

Protection in Hardware:

- Special hardware can run encypted software on secure co-processors. This method is safe, but not cost-effective for widespread use.
- Smart cards can manage secure data, and provide some secure computation.

- Dongles are pieces of hardware that plug into the computer to verify that the particular software package you are running is legitimate. Each dongle-enables software package has its own dongle, and the security can be removed by identifying all of the software-dongle communication.

Protection in Sofware:

- Obfuscation can make the code harder to figure out, thus harder to find the security.
- Self-modifying code, or code that generates other code at run-time.
- Code encryption and decryption, or code the decrypts at run time. This does not scale well to large applications.

3 Guarding Software

The guards are a network of execution units that are embedded in a program. These guards can of course be programmed to do anything, but the authors defined two specific computations: checksumming, and repairing. Each computation is done on small chunks of code which are determined when the guards are installed in the program.

3.1 Checksum Guards

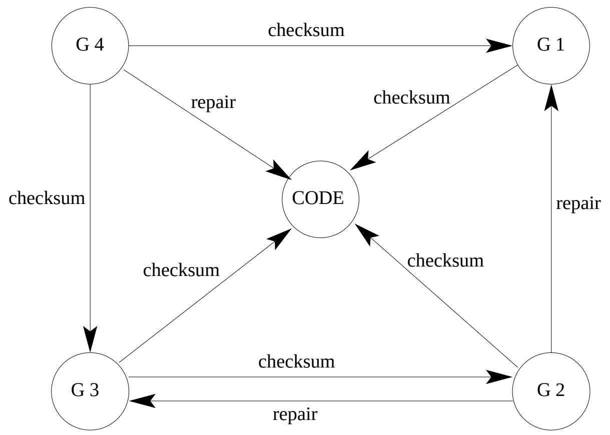
The checksum guards are programmed to check a code fragment within the software. If this code is tampered with in any way the guard takes action. The type of action the guard takes can vary in degree from logging the infraction to halting the program. These guards make the program (in some sense that the authors understand) "self-aware" of its own integrity.

3.2 Repair Guards

The repair guards can fix a tampered section of code. Thus, the changes an attacker made are wiped out and the program remains secure. The repair can be made by storing a clean copy of the segment of code the guard is protection somewhere, and then replacing the dirty copy with the clean one. The authors call this "self-healing."

3.3 Security Through Networks

So far the only difference between guards and a monolithic protection scheme is that there are several blocks of protection code, not just one. The security of the guards actually comes from guards watching each other's backs. Each guard protects fragments of code, and that includes protecting other guards. So, if a *cracker* finds one guard and tries to remove it, she will be unsuccesful if the guard is guarded. Below is an example of a guard network (see the actual paper for a better example):



It is important to note the the guards have to be placed in particular execution orders. A repairing guard must come before the code it fixes, or else it is rendered useless.

This protection scheme is more secure than the monolithic protection scheme because it is distributed. Every guard would have to be identified and simutaneously disabled for the attacker to break it. It also allows for many different guard configurations, so knowing the numbers and types of guards won't necessarily help in removing them. Also it can scale easily to larger programs by simply increasing the size of the guard graph.

4 Implementation

To implement the guard code the authors recommend two attibutes that the guards should have:

- **Stealthiness**. Each guard should be programmed slightly differently so if one guard is detected, the other guards can't be found by simply looking for the same code in a different block. The actions of the guard should be as inconspicuous as possible. So, if a checksum guard decides to halt the program instead of halting immediatley it could set off a chain of events that halts the program several thousand instructions later with no clear trail leading back to the guard. They also recommend bluring the boundries of run-time data and executable areas of the program by inserting executable code in the data blocks.
- **Tamper-resistance**. Obfuscating the guard code would make it harder to detect and harder to understand if found. The authors recommend some techniques defined by Christian Collberg and others.

Apparently the authors have implemented all of their own suggestions. They built a program that inserts a guard network in Win32 executables. The program modifies the binaries and inserts a user-specified number of guards. No two guards are exactly the same to achieve stealthiness, but they have the same set of functionality. The program creates the network of guards, and automaticly inserts them in the binary.

4.1 Experimental Results

The authors found in some cases that inserting guards into the binaries did not increase the executable size, while increasing the number of instructions. This must be some mysterious property of Win32 executables. The average guard size did increase with the number of guards inserted, and while this was not directly explained it could be due to each guard has slightly different code, and the more code you have the more ways there are to permutate it.

The guards did slow down the performance of the software it was tested on. The slow-down varied, but was as much as 32.2% for the program `disasm` with 70 guards installed. The authors discovered that severe slow-down occurs when guard code is inserted in highly repetitive loops. If each guard only executed once the slow-down only went as high as 4.9%, also for the program `disasm` with 70 guards. In a typical run of the programs they tested the guards on only between 7% and 75% of the guards installed in the software were actually executed. The authors added an option in the guard-insertion application where the user can highlight areas where they recommend guard insertion to improve performance.

References

Hoi Chang, and Mikhail Atallah. *Protecting Software Code by Guards*