Seminar "Malware"
Dipl.-Inform.Daniel Loebenberger
Bonn-Aachen International Center for Information Technology Winter 2007/08

Equo ne credite: Trojans

Author: Daniel Rosenthal

Date: December 16, 2007

Contents

1 Some examples of Trojan Horses and a little bit of history 2
1.1 Greek mythology . . . 2
1.2 Back Orifice . . . 2
1.3 InCommand . . . 2

2 Definition of a Trojan Horse 2
2.1 Characterization . . . 3

3 Trojan Horses in Compilers 4
3.1 Self reproducing by substitution . . . 4
3.2 Conditional self reproducing . . . 4
3.3 Bootstrap Test . . . 5
3.4 Passing the Bootstrap Test . . . 6
3.5 Avoiding Trojan Horses in compilers . . . 7

4 Hunting Trojan Horses 7
4.1 Harrier as part of HTH . . . 7
4.1.1 Abstraction levels . . . 7
4.1.2 Security policy . . . 8

5 Summary 8

Abstract

Probably you already know this scenario: In your e-mail box is a message from a friend with an attached greeting card. You open this greeting card and it shows a nice picture of your friend in holiday. Some hours/-days later, your computer starts to do courious things. The DVD-ROM Drives opens or the screen gets blank, but you are sure you did nothing to cause this. Perhaps you activated a Trojan Horse by opening the e-mail attachment containing the greeting card from the friend mentioned before. Hopefully he just wants to play a trick on you and did not do harmful things to your computer or your personal data stored on the harddrive.
This paper deals with Trojan Horses, Malicious Software that can do many harmful things to a computer, once executed. We will see some examples of Trojan Horses to get a better understanding of what they are doing and how they work and a kind of historical overview. After that, we will formulate a precise defition and build up a characterisation of Trojan Horses. Later on, we will see how Trojan Horses in binary compiler implementations can pass nearly every state of the art compiler test, without beeing noticed. The last but one part will deal with a security framework to detect new Trojan Horses online at runtime and have a detailed look on the security monitor beeing used. The last part actually will sum up this paper.

1 Some examples of Trojan Horses and a little bit of history

1.1 Greek mythology

The name Trojan Horse comes from the greek mythology. After ten years of siege in the trojan war, the greek used a trick to win the battle against the Trojans, because they can not break the walls of troy, nor the gates. The greek built a huge wooden horse and inwhich their best soldiers were hidden. They said to the Trojans : "We can't defeat you within ten years and leave the battlefield now and give you this horse as a present." The greek army left, but not so far away they waited until the night. In between the Trojans pulled the horse inside the city and made a big party and everyone got drunk. At night, when no Trojan was awake, the hidden greek soldiers left the horse, opend the gate for the rest of the arriving army and so the Greek entered Troy and defeated the Trojans.

1.2 Back Orifice

This Remote Administration Trojan Horse (RAT) was developed by a group called "Cult of the dead cow" and published at DefCon 6 (a yearly conference of hackers in Las Vegas) in 1998. It affects Windows 95 and 98 operating systems and is a modularized system of plugins. So one can add a new functionality or feature easily by adding a new plugin.
After executing this program by the unsuspected victim, it sends his machines IP-adress and portnummer via ICQ or IRC to the attacker. Then the attacker is able to build up a connection to the victims computer and send commands to the Back Orifice Trojan Horse. These commands are executed and any information requested by the attacker will be send. As an example he can copy the victims entire harddrive or maybe even manipulate the data on this way. After doing his malicious things the attacker can send a clean up command that will remove all files and registry entries of Back Orifice, so that no one can decide whether Back Orifice was ever installed to the computer or not.

1.3 InCommand

InCommand is (like Back Orifice) a RAT affecting windows operating systems, containing a server part running on the victims machine and a client part, running on the attackers computer. With this interface the attacker can for example start an ftp server on the victims machine, or open his DVD-ROM drive, or turn off his monitor. More options you can find on Figure 1.

2 Definition of a Trojan Horse

In the last section we have seen some examples for Trojan Horses and got an understanding for what they are used for. Now we formulate a more precise definition.

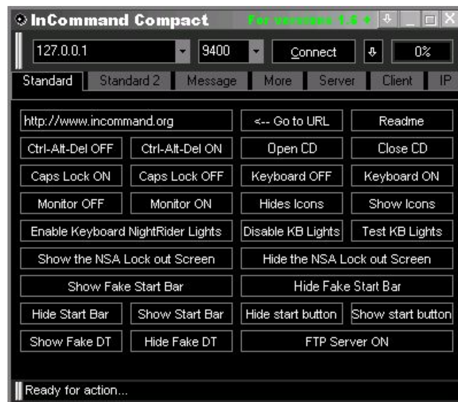

Figure 1: InCommand

Definition 1 (Trojan Horse) *"An apparently useful and innocent program containing additional hidden code which allows the unauthorized collection, exploitation, falsification, or destruction of data."[6]*

We see that in contrast to viruses Trojan Horses must not have a self reproduction method included. They must be run by the user, so the Trojan Horse program has the same privileges as the user. With administrator or root rights, there is no limit to the installation on a computer. You see, it is dangerous always to login as "Administrator" on a Windows machine to do every day tasks and maybe execute accidently a Trojan Horse.

2.1 Characterization

To distinguish different types of Trojan Horses we will now build up a characterization. We distinguish between: [1]

- Propagation Methods **P**
- Activation **A**
- Placement **H**
- Effectiveness **E**
- Communication **C**
- Functions **F**
- Guarding Mechanisms **G**

This allows us to describe every Trojan Horse as a tupel

$$T = (P|G, A|G, H|G, E|G, C|G, F|G) \tag{1}$$

where the Guarding Mechanism for every item can vary.

3 Trojan Horses in Compilers

Ken Thompson, the inventor of Unix, demonstrated on his Turing Award Lecture in 1984, how Trojan Horses can settle down in binary compiler implementations without beeing noticed by every state of the art compiler validation. For example like the Bootstrap Test, or any amount of source code inspection and verification. Thompson's Trojan Horse was a C compiler, where the Trojan Horse was not visible in the compiler source code, but it was reproducing itself (with Trojan Horse) applied to the compiler source code. Compiling `login.c` intudes a backdoor into the Unix "login" command. How is this possible? The Trojan Horse is not in the source code! Actually, the Trojan Horse was not in the source code what Thompson showed. We will now see how this is done.

3.1 Self reproducing by substitution

As a starter consider the following c program:

Listing 1: self reproducing program (by substitution) [3]

```
main(){
  char *b = "main(){
    char *b = %c%s%c;
    printf(b,34,b,34);
    }";
  printf(b,34,b,34);
}
```

This program prints out it's own source code. How is it done ? The %c means replacing a single character and %s an entire string. 34 is the ASCII value of ". The printf command prints out the string b and as additional parameters, replaces the next %c by 34("),%s by the string b itself and %c by 34 again.

3.2 Conditional self reproducing

Now we consider a program that only reproduces, if a special condition is fullfilled.

Listing 2: Conditional self reproducing[3]

```
//file: reproduce.c
char *buf ="
  //file: reproduce.c
  char *buf = %c%s%c;
  int main(int argc, char *argv[]){
    if (argv[1] && (strcmp(argv[1], %cident%c) == 0))
      printf(buf,34,buf,34,34,34,34,34,34,34);
    else if ((argv[1] && (strcmp(argv[1], %clogin%c) == 0))
      printf(%cOops%c);
    else
      printf(argv[1]);
  }
  void cheat () {}
";

int main(int argc, char *argv[]){
```

```
  if (argv[1] && (strcmp(argv[1], "ident") == 0))
    printf(buf,34,buf,34,34,34,34,34,34,34);
  else if ((argv[1] && (strcmp(argv[1], "login") == 0))
    printf("Oops");
  else
    printf(argv[1]);
}
void cheat () {}
```

In this program, the same technique as in the above example is used, but only if the argument is "ident". In case of "login" it will cause a catastrophe ("Oops") and in any other case, it just prints out the argument.

3.3 Bootstrap Test

Compiler bootstrapping means compiling a compiler, where the source language and the implementation language are the same. For example if you use an C++ compiler to compile a new version of it and the sourcecode for this is written in C++, then you call it compiler bootstrapping.
Now Consider Figure 2.

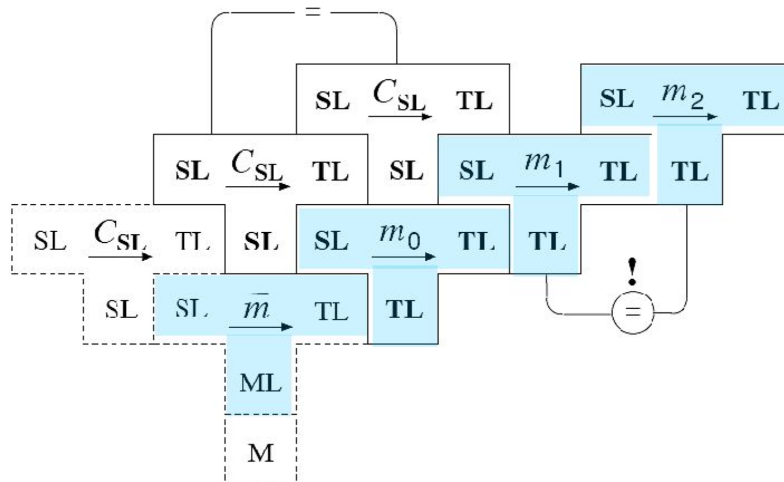

Figure 2: Bootstrap Test

This shows the so called Bootstrap Test. Every T-shaped box is a compiler program, getting it's Input from the left hand side of the box, and outputs on the right hand side. Compiling a compiler returns a compiler. If you repeat this several times, you can play a domino game with these boxes.
C_{SL} is the compiler source program, $\bar{m}$ an existing compiler program and ML is M's machine language. Every compiler compiles an SL lanuage code into a TL language. Suppose we use $\bar{m}$ to generate an initial implementation m_0 in TL out of an SL code. If it works we take m_0, input SL and generate m_1. We don't trust m_0, because it was generated by the unknown compiler $\bar{m}$, but now m_1 is a TL program generated according to C_{SL}. Now we can use m_1 to generate

m_2. m_2 and m_1 will be equal, iff we did not change the C_{SL} in between. With these observations we can now formulate the following two Theorems:

Theorem 1 (Bootstrapping Theorem) *"If m_0 and C_{SL} are both correct, if m_0, applied to C_{SL}, terminates with regular result m_1, and if the underlying hardware worked correctly, then m_1 is correct."[3]*

Theorem 2 (Bootstrap Test Theorem) *"If m_0 and C_{SL} are both correct and deterministic, if m_0, applied to C_{SL}, terminates with regular result m_1, if m_1, applied to C_{SL}, terminates with regular result m_2, and if the underlying hardware worked correctly, then $m_1 = m_2$."[3]*

3.4 Passing the Bootstrap Test

So, if we have an compiler, we can verify that it reproduces itself correctly and we can have a look at the source code and see, that there is no Trojan Horse in it. But how did Thompson's modified C compiler pass this test ? Remeber the conditional self reproducing program in listing 2. Thompson used this technique and he used the fact that we are using compilers ! So consider the next listing:

Listing 3: Passing the Bootstrap Test [3]

```
//file: compile-incorrect.c
#include<string.h>
#include<stdio.h>

char login[255] = "void main() {printf\"Cheating Login (Oops):\";}";
char *buf = "...
";
char cmdbuf[255] = "make CC=gcc 'basename ";
FILE* handle;

void main (int argc, char *argv[]) {
  if (argv[1] && (strcmp(argv[1],"compile.c") == 0)) {
    system("mv compile.c .compile.c.orig");
    if ((handle = fopen("compile.c", "w+"))!= NULL {
      fprint(handle,buf,34,92,34,92,34,34,34,buf,34,34,...,34);
      fclose(handle);
      system("make CC=gcc compile");
      system("mv .compile.c.orig compile.c");
    };
  }
  else if (argv[1] && (strcmp(argv[1],"login.c") == 0)) {
    system("mv login.c .login.c.orig");
    if ((handle = fopen("login.c", "w+"))!= NULL {
      fprint(handle,login);
      fclose(handle);
      system("make CC=gcc login");
      system("mv .login.c.orig login.c");
      };
  }
  else {
    strcat(cmdbufm argv[1]); strcat(cmdbuf," .c'");
  }
}
```

If you consider this program as an compiler, it compiles every source correct, except "login.c", where it compiles a bug (the catastrophe) into and if you want to compile "it's" source code "compile.c" (the non-infected), then it will compile the infected source code and restore the original non-infected source code. This you can repeat as many times as you like and so it will pass the Bootstrap Test.

3.5 Avoiding Trojan Horses in compilers

As we have seen, source level verification and syntactical code inspection does not work. What we need is also semantically correctness. This can we achieve by introducing a semantical correct compiling relation $CC_{SL,TL}$ beween the source and target language. Let C_{SL} be a correct refinement of $CC_{SL,TL}$. If m is applied to C_{SL} then $C_{SL} \in CC_{SL,TL}$. It follows that C_{SL} is a correct implementation of $CC_{SL,TL}$. That implies that m is a correct compiler executable from SL to TL.

4 Hunting Trojan Horses

Zero Day attacks and new malicious code can go undetected by even the most up-to-date antiviral software. Some Trojan Horses executes as plugins to other programs or as DLL (Dynamic Link Library) and may have very little impact on the system behavior. So it is very difficult for the user to detect and may be undeteced for a long time, where the system is vulnerable.
The security framework Hunting Trojan Horses (for short HTH) is intendend to be a complement to antiviral software and developed for detecting difficult types of intrusions.[4]

4.1 Harrier as part of HTH

Harrier is an application security monitor and the heart of HTH. The runtime monitor collects dynamically execution related data across different abstraction levels. Harrier allows the identification of abnormal program behaviour and enables defending againt harmful activities. It uses no source code analyzing, but works with program binaries (only for Linux at this time). The monitoring is restricted to only a few shared objects with a defined API, to handle this huge flow of information and to be able to "use" the computer at the same time. Actually it slowed the system down roughly by a factor seven.

4.1.1 Abstraction levels

Harrier uses three abstraction levels to collect information about the program semantics and the program information flow (see also Figure 3):

1. Architectural (ISA)
2. Operating System (API)
3. Library (API)

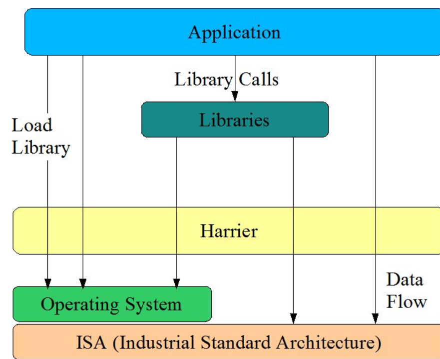

Figure 3: Harrier events

4.1.2 Security policy

The rules beeing used as security policy are classified by three types: The *execution flow* are rules that monitor execution and invocation of new processes in order to detect malicious code being executed. There are also rules that monitor the number of new processes and the rate of creation of these to track *resource abuse*. The last set of rules enforce the *Information flow* between different sources and targets for the different resource types (user input, file, socket, binary, hardware).

5 Summary

Trojan Horses are programs containing additional hidden code that can be used for unauthorized collection, exploitation, falsification, or destruction of data.
We have seen, that if a Trojan Horse is injected in an binary compiler implementation, that source level verification is not sufficient to guarantee compiler corectness. What we need is binary compiler implementation verification.
Harrier within the Hunting Trojan Horse framework is a complement to antiviral software. It's a runtime security monitor analyzing program binaries. It tracks the Architecture, the Operating System and selected library events to detect Zero Day attacks and new malicious code.

References

[1] Dr.-Ing. Claus Vielhauer, Dipl.-Inform.(FH) Andreas Lang: *Lecture slides: Sicherheit verteilter Systeme SS2007*, FH Brandenburg, 2007

[2] Paul A. Karger: *Limiting the Damage Potential of Discretionary Trojan Horses*, in 1987 IEEE Symposium on Security and Privacy, pp. 32-37, 1987

[3] Wolfgang Goerigk: *On Trojan Horses in Compiler Implementations*, in Proceedings of the Workshop Sicherheit und Zuverlssigkeit softwarebasierter Systeme, IsTec Report, IsTec-A-367, Garching 1999

[4] M. Moffie, W. Cheng, D. Kaeli, Q. Zao: *Hunting Trojan Horses*, AsiD'06: Proceedings of the 1st Workshop on Architectural and System support for improving software dependability, pp. 12-17, San Jose, California, 2006

[5] A. Brown, T. Cocks, K. Swampillai: Spyware and Trojan horses, Seminar on Computer Security, 01-04 2004, University of Birmingham

[6] Texas State Library and Archives Commission, http://www.tsl.state.tx.us/ld/pubs/compsecurity/glossary.html, last visited: 2007-12-11