Software Cracking (April 2007)

Ankit Jain, Jason Kuo, Jordan Soet, Brian Tse

***Abstract*—The goal of this project was to analyze existing cracking counter measures within readily available software on today's market. By cracking the software with different techniques we found that most software on today's market employ fairly rudimentary security schemes that are easily by passed with very little know how.**

***Index Terms*—Software Cracking**

I. Introduction

In today's world piracy accounts for $35 billion in lost revenues to software companies. Part of this staggering number is due to the easy crackability as well as rampant distribution of modern software. Bit torrent and peer-2-peer applications have made the distribution of cracks and pirated software as easy as a few clicks. We set out to try to crack popular licensed software with different techniques to show how easy or hard it is to crack software. Even with rampant piracy we found that most software applications implement security measures that are easily by passed. A general trend that appears is software from a large company is usually more secure while smaller start-up companies seem to lack the necessary means to protect their software. To test the pieces of software we utilized three different techniques: Hex Reading, Hex Editing, and Debugging. Using these techniques we found popular applications that were easily cracked. To mediate these problems there are several ways to prevent or increase security, developers can implement encryption so that keys are not visible as plain text. Other techniques include checking for debugging software DLLs, if one is found then the application halts execution. After our experimentation we found that confidentiality and integrity were not upheld in most of today's software. Confidentiality and integrity were breached since a user could easily read or modify the data. In this report we will investigate the several cracking techniques and suggest ways to prevent or mitigate these problems.

II. Cracking Tools

To aid us in our cracking we utilized several different tools that are widely available online. These tools are they key to successfully cracking a piece of software and some have high learning curves to master.

A. Softice

Softice is a kernel level debugger that is capable of halting all instructions in windows. The most useful aspect of Softice is its ability to step through code while an external application operates. For example Softice can detect the lines of code where you enter in an invalid registration code and a message window informing you of this appears. Knowing this information is crucial as it allows a cracker to jump to those lines of code and augment them in such a way to disable or skip the built in security.

B. WDASM32

WDASM32 is a disassembler, which basically takes machine language and translates it to assembly language, much like how an assembler takes assembly and translates it to machine code. This is extremely important for cracking as it allows you to view a program's code line by line. This can be useful since some applications calculate serial keys within the code, and if the algorithm is visible it's possible to replicate it to generate a new serial key.

C. Hiew

Hiew is a hex-editor that allows a user to change hex values for a given application. Doing so enables a cracker to modify key lines of code. For example a user may replace a jump command with a no-op command thus rendering the jump useless. This may be useful when an application displays a warning window telling the user that the serial-key entered is invalid. By nullifying this, a user may skip the message box and register for an application unhindered.

D. RegMon

RegMon is a system administration tool that lets you observe all actions attempted against the windows registry. For cracking, this may be useful as a serial key may be stored in the registry and realizing that an application is accessing that may be crucial.

E. FileMon

FileMon is similar to RegMon however instead of observing the registry, it observes all accessed files. Again this may be useful since the application may be accessing algorithms or serial keys from a separate file.

III. CRACKING EXAMPLES

To gain a greater understanding of software cracking we attempted to crack a number of commercially available software products. This experience allowed us to use the tools mentioned previously to see firsthand what kind of mechanisms most types of software employ to prevent software cracking. It should also be noted that although the steps described here for cracking the software are given as if there is a certain algorithm to follow, in actuality there was a lot of guesswork involved to figure out the steps necessary to crack the software.

A. *Hex Reading*

To start out we will give a simple example of using hex reading to crack software. We will crack the program BackupDVD. This technique is quite simple, and is almost trivial, but it still works in some cases and so it is worth using it as an introduction.

1) *Step 1: Examine the security measures*

The first step in trying to crack a program is to examine what kind of protection it uses. Opening BackupDVD we see that it gives you the opportunity to register the program using a serial key. If you enter the wrong serial key, a message pops up saying "Invalid Serial Key, try again!" (see Fig. 1). We will make a note of this message, as we will need to remember it later.

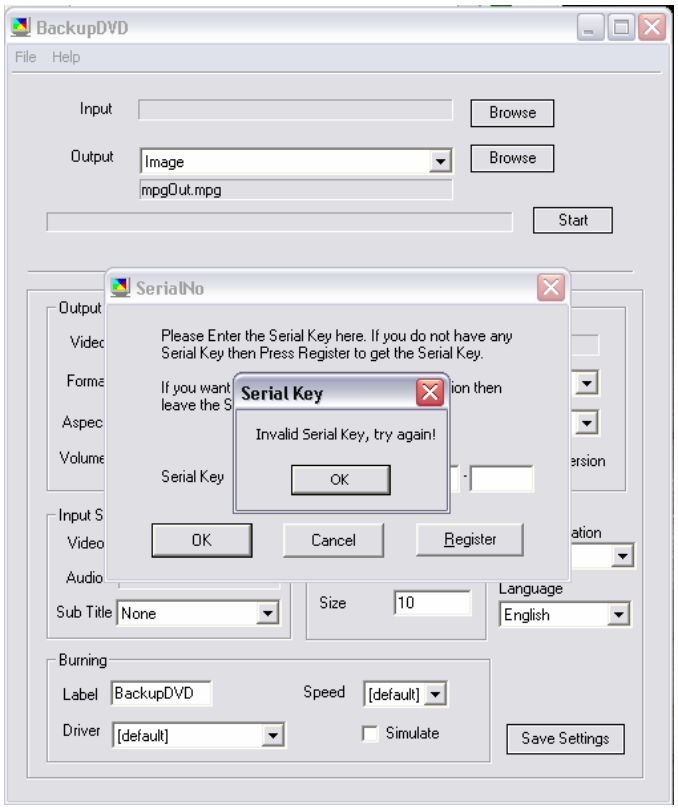

Fig. 1. BackupDVD, showing the message box which pops up when an invalid serial key is entered.

2) *Step 2: Examine the program*

After this we can open the folder for BackupDVD. Here we may notice that in addition to the BackupDVD.exe executable which runs the main program there is also an executable called BackupDVDSK.exe. Running this we find out that it is a separate program responsible just for the serial key registration. We can now open up this program in a hex reader and search for the message we noted earlier, "Invalid Serial Key, try again!" Examining the area around this message, we see that there is a string nearby which looks like a possible serial key (see Fig. 2). Entering this key into the registration, we see that it works and the program is now registered!

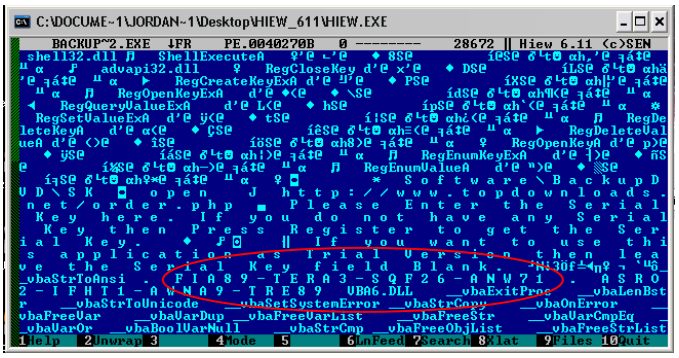

Fig. 2. The serial key being shown in plain text when the program is viewed with a hex editor.

B. *Hex Editing*

Unfortunately most programs do not use such simple security measures and so more sophisticated techniques are needed. In this example we will crack the popular file archiving program, WinRAR. It will use similar principles to try and find the area of the program where registration takes place.

1) *Step 1: Examine the security measures*

Once again the first step is to examine the security measures. Similarl to the last program we see that we must enter a name and a serial key. Then if we enter the key wrong a message box pops up saying "Registration Failed". Again we write down the message which is in the message box for later.

2) *Step 2: Examine the program*

Unfortunately this time the program is not as simple so we will open it in a disassembler to get a more complete picture of what is happening. Once the program is disassembled we can search for string references to strings used inside the program. Searching through these it is easy to find the one from the message box that we wrote down earlier. By clicking on this reference we are brought to the area of code where the message box is created (see Fig. 4). Examining around this area we can see that there is a compare instruction followed by a jump if not equal instruction just above where the message is used (see Fig. 4). It is very likely that this instruction represents the serial number that we entered being compared to an internally generated serial number and then jumping if it is correct, so we will make a note of the offset here.

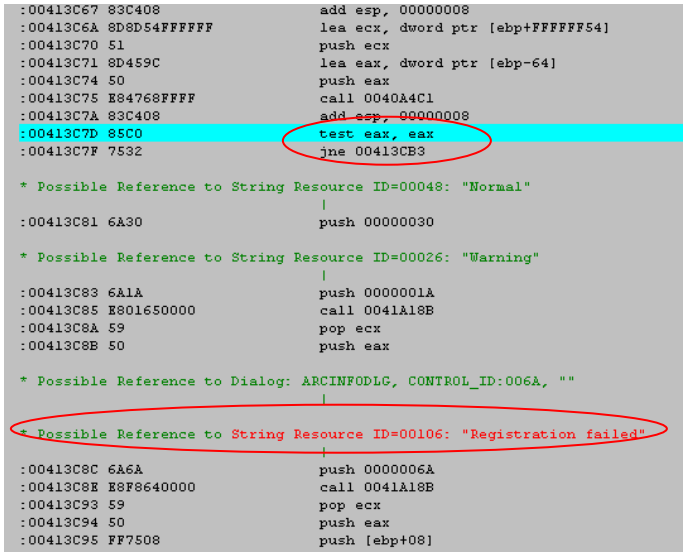

Fig. 4. The reference to the string and the comparison and jump prior to the message.

3) *Step 3: Hex editing*

We will now open the program in the hex editor and change the viewing mode to assembler. We then jump to the offset that we wrote down and this will bring us to the jump instruction that we were looking at. We want to make it so that this jump will be taken no matter what, so we instead replace the jump if not equal instruction with a jump instruction (see Fig. 5).

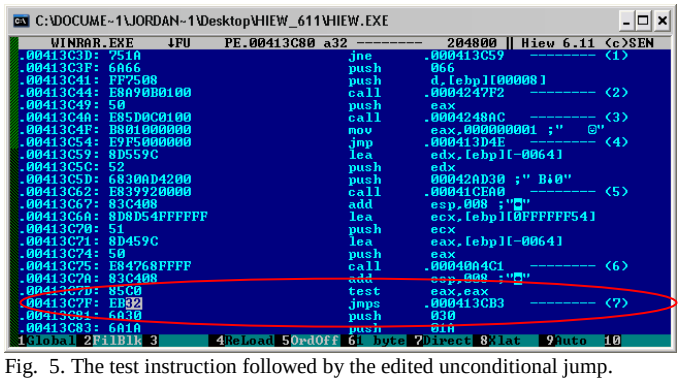

Fig. 5. The test instruction followed by the edited unconditional jump.

We then save the changes to the program and exit the hex editor. We now try registering the program again with some random input and this time we see that it allows us to register (see Fig. 6).

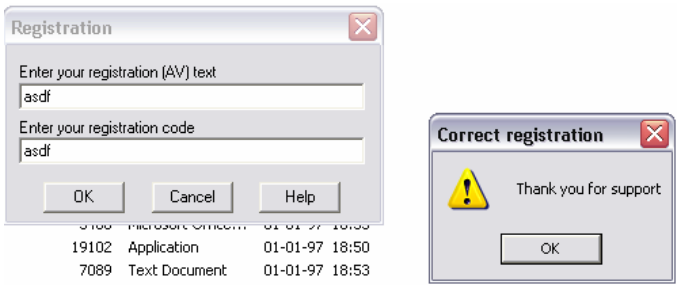

Fig. 6. WinRAR showing that we have registered using invalid serial keys.

C. *Debugging*

Unfortunately sometimes string references cannot be found in the program as they are not hard coded in and are instead accessed through another file, or other methods. In this case what you can do is use a debugger such as Softice to set a breakpoint in the program on a Windows API call which you can guess will be used in the program. Some examples include the calls messageboxa, readfile, writefile, regcreatekeya and many others. Once you have found a call which is used and the breakpoint happens, you can find the area of code where it is called and similarly to hex editing, find a possible compare and jump instruction and edit it so that it thinks you have correctly entered the key. In this example we will crack the popular internet chat client, mIRC.

1) *Step 1: Examine the security measures*

Once again, similar to the last two examples, we examine the program to see what kind of security measures in place. Again, it uses a name and serial key and we make a note of the message which pops up when we enter the incorrect key. Unfortunately, when we examine the program this time we find out that we can't access the string references so we are forced to use a debugger instead.

2) *Step 2: Setting breakpoints using the debugger*

Using the Softice debugger we set a debugger on the messageboxa API call, since a message box is popped up by the program when you enter the wrong serial key. Once the breakpoint is set using the command "bpx messageboxa" we again enter an incorrect serial key and just before the message box pops up the Softice debugger appears since this is where we set our breakpoint. When the debugger pops up we press F10 to step out of the actual WindowsAPI call and back to the program code. Once here we make note of the line number that we are at and then erase the breakpoint and then erase the breakpoint and close the debugger.

3) *Step 3: Examine the program*

Once we know where to look we can open the program in a disassembler to examine it in more detail. Examining the area around where the message box is created we see that it is referenced by a jump at a previous point in the program (see Fig. 7). Going to this point we see that again there is a comparison followed by a jump if equal. Once again we make a note of the offset here.

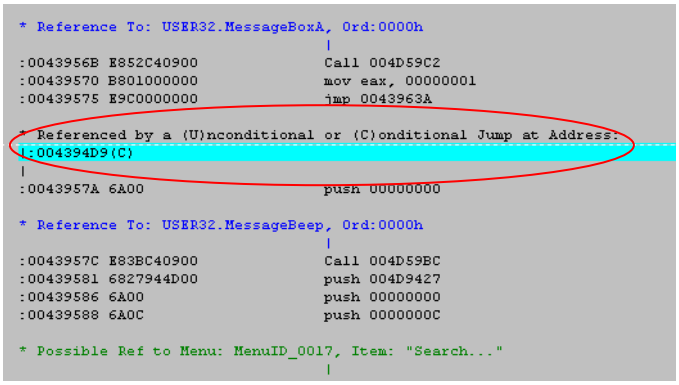

Fig. 6. The jump from a previous point in the program.

4) Step 4: Hex editing

Opening the program in a hex editor we can go to the offset we found. This time we want to make sure that the jump is not taken, so we will replace the jump with a series of no operations. We need to make sure that the rest of the instructions are not altered though, and so since the jump if equal instruction is 6 bytes long including its arguments and the no operation instruction is only one byte, we need to replace it with 6 no operations (see Fig. 8).

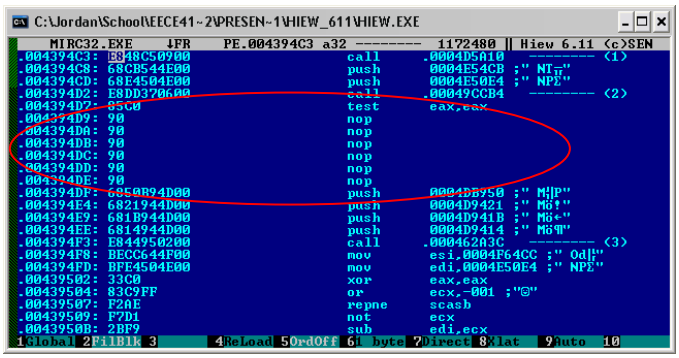

Fig. 7. The edited program with the jump if equal instruction replaced by 6 no operations.

Once this is done we can run the program and see that registration works fine now (see Fig. 9).

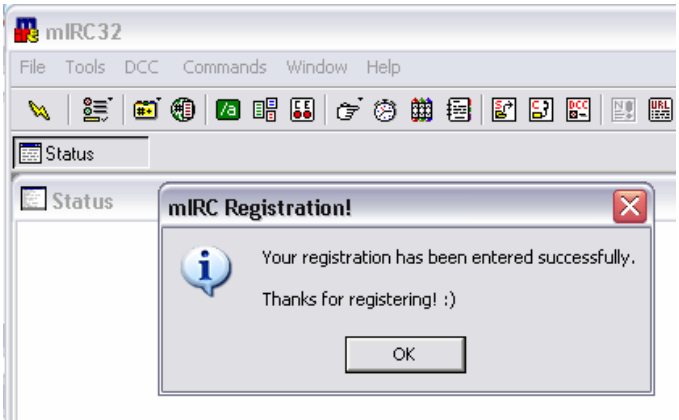

Fig. 8. The mIRC registration showing that we have properly registered after entering an invalid serial key

D. Software design principles ignored

Obviously with these examples there were numerous design principles ignored but the main ones ignored were the following:

- Open Design: This is mainly for BackupDVD. There was the assumption made that the internal code would not be visible to the users of the program and so it would be ok to hardcode the serial key. If the design was open this assumption wouldn't have been made. This is also applicable to the others though, since they weren't designed openly and so they do all partly rely on security through obscurity.
- Defense in Depth: In all these examples there is only one software check done to check that the user has registered.
- Question Assumptions: Again, with BackupDVD it should not have been assumed that users would not be able to access the internal workings of the program.
- Complete Mediation: All the programs should check if the user is registered every time a feature is used, whereas instead it is just checked once and then a flag is set to say that it is registered.
- Separation of Privilege: In each of the programs only one condition is used to check the privilege of the user, instead of using multiple conditions.

IV. Countermeasures/Prevention

Having analyzed the different ways one could get around CD-key or even generate it to illegally access a piece of software, we should now try to define some countermeasures that can help software developers safeguard the integrity and confidentiality of software.

A. Encryption

Giving the example of cracking a single master key that is hardcoded into the program, it is observed that a complex hiding mechanism must be used, namely, encryption. Encryption has become the most common security measure in the digital world. Almost all communications involves some sort of encryption algorithms. Thus, by integrating encryption in the codes to encrypt the serial keys, messages, or functions, keyword searching or hex viewing cracking methods will become more difficult to crackers. Online sources suggested that a simple XOR adds 5 minutes to cracking time, while more advanced RSA/SHA encryption standard can add hours.

B. Detection

When a debugger is used, it will create a process running in the background of an OS. Debugger like Softice run a driver file called NTICE.dll while continuously monitor the OS. One way to detect such file in the kernel is to create a virtual file with the same name. Using Application Programming Interface (API) function "CreateFileA", we can detect whether NTICE.dll is

running. Then the developers and program in such a way to have the software terminates itself when debuggers are detected.

An example of such usage is documented below:

```
HANDLE hFile = CreateFile( "\\\\.\\NTICE",
GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);

if(hFile!=INVALID_HANDLE_VALUE)
{ // Softice Detected
CloseHandle(hFile); }
```

C. Selection

Selection means that software developers must choose the proper programming tools and security software that are available out there to decide how much security they need. For example, different sources have indicated that Visual Basic and Dephium are two languages that are much harder to crack than others, simply because their run time .dll files are harder to decipher, making isolation of software protection more difficult.

Furthermore, there are numerous software released out in the market for the benefit of software developer so they can avoid having to develop security algorithms on their own. One risk of using these public known software is that they might be malicious themselves. One of the concerns is the use of backdoor. The protection software might have unknown functionalities that serve the purpose of snooping CD-serials and encryption algorithms. Another concern is that the protection software might already be cracked itself. Using it means leaving your own software vulnerable to crackers everywhere.

Therefore, depending on how much security is needed, the developer must choose the appropriate programming language and protection software through research.

D. Experimentation

Although cracking skills and experiences are not common amongst developers, one of the best defensive mechanisms is to identify the weakness of your own protection scheme by trying to crack your own software.

There are web crackers who offer help to improve protection scheme, for those who would like to gain an understanding how much a piece of software can resist attack. Proper use of such service and attempting to break your own code might be the best option out there to improve

E. Other protection scheme

1) Fake serials

Sometimes when the first set of serial is found, crackers will stop at that point. Because of this human nature, one could implement a fake serial that is relative easy to find and work for registration at the same time. Every time this fake serial is used in registration, the program removes some of the full version functions.

2) Online license

Localized software is vulnerable to attack because the users have all the world's time to attempt cracking. Online registration and license files synchronization eliminates such problem by requiring users to obtain serial key and license file every time the software is used. In this case, crackers are isolated. Online server hacking becomes the primary threat.

V. COUNTER-COUNTERMEASURES

The truth about the software world is that crackers are always one step ahead of software developers. No matter what protection scheme is used, crackers can always find a way to snoop out the registration weakness, and counter any pre-cautions developers take.

For example, in the case of debugger detection, newer debugger versions have included a sort of polymorph function to change its running driver name. Whenever the debugger is required to run, the file NTICE.dll will be appended with the debugger serials.

NTICE.dll ➔ NTICE1234.dll

Because of this method, the detection scheme would fail.

VI. TOO MUCH SECURITY:

Now we discuss some examples of software that are designed to provide security but rather end up serving some other purpose.

For example, Starforce is very famous anti-cracking software that many gaming companies used to provide security from hacking their games. Now this software comes as a driver hidden in the game and it installs itself as a disguised programming running on a PC. It does provide security against hacking, but it also slows down the computer to an extreme extent, blocks the CD/DVD ROM from operating and, promptly keeps on restarting the computer again and again. This software is boycott on many online blogs and considered as malware on the Internet.

VII. SOFTWARE GUARDS:

Now we discuss the software guards approach to secure a network. Most of the software based mechanisms to protect software are either too weak i.e. they have a single node of failure, or they are too expensive i.e. they incur heavy run time penalties on performance. A simple approach to prevent this is to use a distributed approach to protect software. In this mechanism, tamper resistance of program code is achieved not by a single security model but by a network of security units.

This approach is followed by using software guards, where a chain of small guards or security units programmed to do certain tasks work together to enforce security by reinforcing protection for each other by creating mutual-protection. These guards can be programmed as Win32 executables and start executing. This network of guards is harder to defeat because security is shared among all the guards, and each of them is potentially guarded by other guards. The guarding framework could be distributed as follows:

- Guards: protection is pro vided by a network of execution units embedded within a program. Each guard is a piece of code responsible for performing certain security-related actions during program execution like. Examples of some of the program tasks that guards can do are: Checksum code, Repair code etc.
- Guard network: A group of guards working together can provide a more sophisticated security mechanism than that provided by a single guard. For example, if a program has multiple pieces of code whose integrity needs to be protected, then it can deploy multiple check summing guards for protecting the different pieces.
- Security:
  - ✓ Distributedness: There is no single point of entry or exit into or out of the network because its individual components are invoked at different points at runtime
  - ✓ Multiplicity: Multiple guards can be used to secure a single piece of code providing variety of security mechanisms to ensure better security
  - ✓ Dynamism: There are many different ways by which security guards can be programmed. For example, a group of security guards can emphasis I/O aspect of security while another group can focus on malware and so forth

VIII. Conclusion

As we've gone through these examples it probably seems that every security mechanism implemented is later bypassed. So what options are we left with, should we not run security checks on the machine? Basically the answer is yes, one of the problems with software on our current systems is that it is inherently transparent to the user. And given this information about the actions a program is performing as well as enough time and motivation any security mechanisms will eventually be cracked. So what are we supposed to do to solve this? Well, we could run our software only on protected systems which hide this information from the user, but this isn't really a viable solution because we want to be able to run on the systems that our clients are using.

So what can be done then? Well, one of the main reasons most software security systems fail is because of a single point of failure, and one of the best ways of guarding against this is to implement multiple checks in your system. An even better method is to use distributed checks, as this way the code is not running on a local machine, making it much more difficult to crack. Another method to deter crackers is to use software aging. This is when you continually update your software by adding new features, thus making it less and less useful to someone who hasn't properly registered.

So finally, in conclusion, looking back we saw that software cracking is a very important issue, partly because many programs only use very rudimentary protection. However, we also saw that basically, any software protection that relies only on local measures can be cracked. So what we are left with is a trade-off between putting enough security in our software to provide a deterrent against most people while not forgetting about the usability and features of the software that we're trying to write.

References

[1] Pavol Cerven, "Crackproof Your Software: Protect Your Software Against Crackers". Berkley: Publishers Group West, 2002.

[2] CrackZ, "Anti-Debugging & Software Protection Advice". [Online] Available: *http://www.woodmann.com/crackz/Tutorials/Protect.htm*

[3] H. Chang, M.J. Atallah, "Protecting software codes by guards" , CERIAS, Purude University and Arxan Technologies, 2001

[4] Boycott Staforce, Glop.org, 2005, http://www.glop.org/starforce/

[5] JA Whittaker, HH Thompson "How to Break Software Security" Addison Wesley, 2003

[6] Markus Jakobsson, Michael K. Reiter "Discouraging Software Piracy Using Software Aging", CERIAS, Purude University and Arxan Technologies, 2001