Dillon Welch

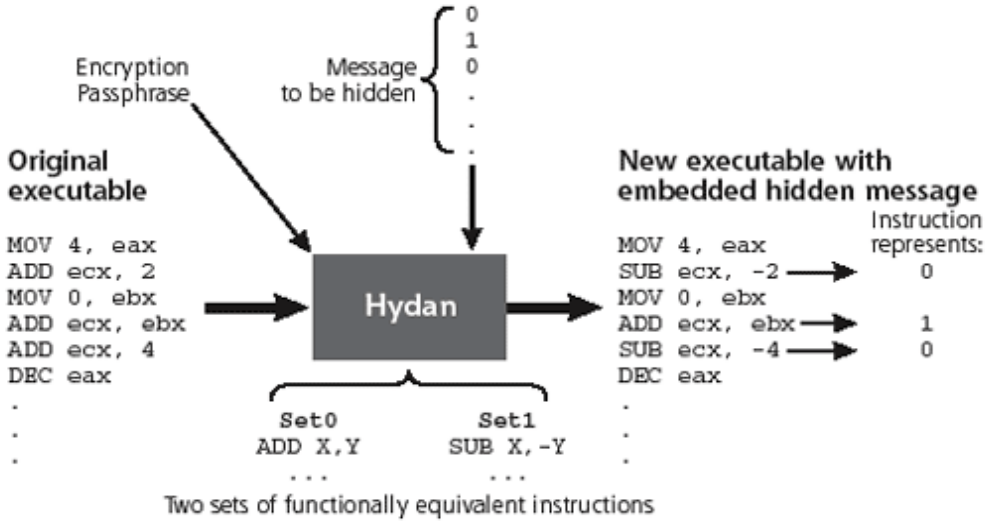
12/13/11

Steganography White Paper

Steganography, coming from the Greek words *steganos* meaning “covered” and *graphei* meaning “writing”, is in general the science of hiding information within other information such that the hidden information can not be read. This concept has been studied since the time of the Greeks, and in recent years has been applied to digital media. Most research in this area has focused on hiding information within audio files and images, as their large size and redundancy allows for large and unnoticeable changes to the file.

The problem becomes more difficult when applied to executable files, as any change to the executable will cause the program to run in a different way, making it obvious that the program has been modified. There has been some research into this field as well, focusing on redundancy in the instruction sets of assembly code. Each choice between two equivalent instructions can be thought of as a bit of data, and this can be expanded for sets of more than two equivalencies. Rakan El-Khalil from Columbia University, in his program Hydan (Old English for the act of hiding something), focused on redundancies in the x86 instruction set. In particular, one of the main redundancies was in adding or subtracting a number. Adding a number is the same as subtracting the negative of that number. Therefore, with the convention of addition representing zero and subtraction representing one, the program goes through and changes the additions and subtractions as necessary to match the message being hidden (See Figure 1). This does not change the way the application runs, nor does it increase the size. Unfortunately, this results in a program with a much larger amount of negative subtractions than the average program that would be easy to detect by statistical analysis. In the paper on Hydan, El-Khalil notes several ways in which the program could be improved. One way is noting that compiled

code is can be inefficient, in particular it could have several additions and subtractions where one would do the same job. Two other ways are using dead code analysis to find blocks of code that can be changed without effecting the program (since they are never reached) and identifying functionally equivalent code blocks such as the order of declared functions. Additional ideas included are changing register allocation choice, changing the order in which elements are pushed and popped off the stack, and changing the order of function arguments.



(Figure 1: Note that this picture uses the convention of subtraction being 0, instead of addition).

Hydan was published in 2003, and the only other research in the same direction I could find so far was an expansion and improvement done two years later on the concepts presented in Hydan. This research goes into greater detail about the various redundancies within an executable, also using x86 as the target instruction set. According to the paper, a compiler goes through four stages, each have a number of redundancies. The first stage, instruction selection, is when code is translated into assembly. Multiple equivalent statements usually exist to implement a given operation. In the second stage, register allocation, there are usually multiple registers that are equivalent to choose from. In the third stage, instruction scheduling, instructions are put in their final order, and there are usually multiple valid orders. In the fourth stage, code layout, there are multiple orderings for the code. They

developed a tool that generates all possible instruction sequences for a given code sequence and output registers. In general, the number of equivalents is exponential to the number of instructions. In practical application, register allocations are unexploitable since the amount of registers is limited and usually most registers are fixed. Instruction scheduling is performed by basic blocks, and so a dependency graph is created for each block, where dependent instructions are connected by directed edges. Applying a branch and bound algorithm to select valid orders of instructions can generate all possible permutations of the instructions. For code layout, they argue that if two blocks of code do not have a fall through path, they can be permuted in any order. They then go into the fact that these methods will also result in unusual code that is easily detectable, just like with Hydan. They discuss several methods that can be used to make the steganography more stealthy, at the cost of data rate.

In addition to this research, I also found a script that hides an .exe within a .gif image. From the description and browsing the script, it seems this is done by creating a new comment block in the .gif with the .exe data and saves it as a .hta.gif file. Then when the file is opened, it opens up the image in IE as regular. When the .gif extension is removed though, it becomes a .hta application, which runs the executable hidden within. This is similar to concept of “dead-code” referred to earlier, as the comment block means nothing unless it is actually executed.

The main goal of any future research is to figure out whether these ideas could be applied to achieve the goal of hiding an executable within another executable such that both will run at the same time. Using the methods outlined, I would think it would already possible to hide the binary of an executable within another executable like a regular message using Hydan, though I would first have to get the code for Hydan to compile. Research could be done into other ways to hide code other than equivalent instruction sets, such as how the script hides an executable within an image. Finally, if the concept of Hydan ends up working, there is much work to be done in improving the methods involved.

In summary, the research I have done shows that work has mostly been done in utilizing redundancies within the x86 instruction set to hide information within executables. There are flaws to this methodology, namely that it is very obvious to the trained onlooker. Methods to execute these algorithms more stealthily have been found, though there is a trade off between stealth and data rate. Also, there has been experimental work done in hiding an executable within a gif image.

References

Anckaert, Bertrand; De Sutter, Bjorn; Chanet, Dominique; De Bosschere; Koen: Steganography for Executables and Code Transformation Signatures. In: Information Security and Cryptology. Volume 3506. (2005).

El-Khalil, R.: Hydan (2003) http://crazyboy.com/hydan/

El-Khalil, R., Keromytis, A.: Hydan: Hiding information in program binaries. In: International Conference on Information and Communications Security, LNCS. Volume 3269. (2004)

Poulsen, Kevin. "Program hides secret messages in executables: Hydan Seek." (2003) http://www.theregister.co.uk/2003/02/24/program_hides_secret_messages/

Skoudis, Ed. "Hiding Data in Executables: Stego and Polymorphism." (2003) http://www.informit.com/articles/article.aspx?p=102181&seqNum=6

"Hiding EXE Data Within GIF Data." http://www.codeproject.com/KB/vbscript/Steganography.aspx