Anti-Disassembly using Cryptographic Hash Functions

John Aycock, Rennie deGraaf, and Michael Jacobson, Jr.
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{aycock,degraaf,jacobs}@cpsc.ucalgary.ca

TR 2005-793-24

Abstract

Computer viruses sometimes employ coding techniques intended to make analysis difficult for anti-virus researchers; techniques to obscure code to impair static code analysis are called *anti-disassembly* techniques. We present a new method of anti-disassembly based on cryptographic hash functions which is portable, hard to analyze, and can be used to target particular computers or users. Furthermore, the obscured code is not available in any analyzable form, even an encrypted form, until it successfully runs. The method's viability has been empirically confirmed. We look at possible countermeasures for the basic anti-disassembly scheme, as well as variants scaled to use massive computational power.

1 Introduction

Computer viruses whose code is designed to impede analysis by anti-virus researchers are referred to as *armored* viruses.[1] Armoring can take different forms, depending on the type of analysis being evaded: dynamic analysis as the viral code runs, or static analysis of the viral code. In this paper, we focus on static analysis.

Static analysis involves the tried-and-true method of studying the code's disassembled listing. *Anti-disassembly* techniques are ones which try to prevent disassembled code from being useful. Code using these techniques will be referred to as *disassembly-resistant code* or simply *resistant code*. Although we are only considering anti-disassembly in the context of computer viruses, some of these techniques have been in use as early as the 1980s to combat software piracy [7].

Ideally, resistant code will not be present in its final form until run time – what can't be seen can't be analyzed. This could involve self-modifying code, which presents

[1]The techniques we describe can be used by any malicious software, or *malware*, so we use the term "computer virus" in this paper without loss of generality.

```
key ← getusername()
hash ← md5(key ⊕ salt)
run ← hash$_{lb...ub}$
goto run
```

Figure 1: Anti-disassembly pseudocode

problems for static analysis of code [8]. It could also involve dynamic code generation such as a just-in-time compiler performs [2].

In the remainder of this paper, we present a new method of anti-disassembly based on dynamic code generation, that has the following properties:

- It can be targeted, so that the resistant code will only run under specific circumstances. We use the current username as a key for our running example, but any value available to the resistant code (or combinations thereof) with a large domain is suitable, like a machine's domain name. Because this key is derived from the target environment, it may be thought of as environmental key generation [10].
- The dynamically generated code is not available in any form, even an encrypted one, where it can be subjected to analysis until the resistant code runs on the targeted machine.
- Even if the dynamically generated code were somehow known or guessed, the exact key used by the resistant code is not revealed.
- It does not rely on architecture-specific trickery and is portable to any platform.

In the remainder of this paper, we explain our anti-disassembly technique and present some empirical results. We then look at how the technique might be countered, along with some more entrepreneurial means of deployment.

2 The Idea

A cryptographic hash function is one that maps each input to a fixed-length output value, such that it is not computationally feasible to reverse the process, nor is it easy to locate two inputs with the same output [12]. Like regular hash functions, a cryptographic hash function is many-to-one.

Our idea for anti-disassembly is to combine a key – here, we use the current username for concreteness – with a salt value, and feed the result as input into a cryptographic hash function. The hash function produces a sequence of bytes, from which we extract a subsequence, and interpret that subsequence as machine code. We will refer to this subsequence as a *run*.

A pseudocode example of this idea is shown in Figure 1. It uses the username as a key, and MD5 as the cryptographic hash function [11]; $\oplus$ is the concatenation operator. MD5 is now known to be vulnerable to collisions [14], but this is irrelevant to our technique: the analyst does not have the MD5 hash, and in any case would be interested in the exact value of the key, not just a key which produces the same hash. In addition, our anti-disassembly technique can be used with any cryptographic hash function, so a different/stronger one can be chosen if necessary.

There are three issues to consider:

1. Having the wrong key. Obviously, if the wrong key value is used, then the run is unlikely to consist of useful code. The resistant code could simply try to run it anyway, and possibly crash; this behavior is not out of the question for viruses. Another approach would catch all of the exceptions that might be raised by a bad run, so that an obvious crash is averted. A more sophisticated scheme could check the run's validity using a checksum (or re-using the cryptographic hash function), but this would give extra information to a code analyst.

2. Choosing the salt. This is the most critical aspect; we suggest a straightforward brute-force search through possible salt values. Normally, conducting a brute-force attack against a cryptographic hash function to find a particular hash value, i.e., a collision, would be out of the question because the hash functions are designed to make this computationally prohibitive. However, we are only interested in finding a subsequence of bytes in the hash value, so our task is easier.

   Approximately half of the output bits of a cryptographic hash function change with each bit changed in the input [12]; effectively, we may consider the hash function's output to change randomly as the salt is changed. Given that, the probability of finding a particular b-bit run in an n-bit output is the ratio of the bits not in the run to the total number of bits: $2^{n-b}/2^n$, or $1/2^b$. The expected number of attempts would then be 2^{b-1}. Furthermore, because only the salt is being changed in the brute-force search, this implies that we would need $b - 1$ bits of salt for a b-bit run.

3. Choosing lb and ub. These values are derived directly from the hash value, once the desired salt is found.

The salt search is by far the most time-consuming operation, but this need only be done once, prior to the release of the resistant code. The search time can be further reduced in three ways. First, the search can be easily distributed across multiple machines, each machine checking a separate part of the search space. Second, faster machines can be used. Third, the search can be extended to equivalent code sequences, which can either be supplied manually or generated automatically [6, 9]; since multiple patterns can be searched for in linear time [1], this does not add to the overall time complexity of the salt search.

3 Empirical Results

To demonstrate the feasibility of this anti-disassembly technique, we searched for the run (in base 16)

```
e9 74 56 34 12
```

Algorithm	*Key*	*Salt*	*Search Time (s)*
MD5 (128 bits)	aycock	55b7d9ea16	39615
	degraaf	a1ddfc1910	47650
	foo	e6500e0214	60185
	jacobs	9ac1848109	28723
	ucalgary.ca	4d21abe205	18220
	Average		44746
SHA-1 (160 bits)	aycock	07e9717a09	36584
	degraaf	0d928a260e	55424
	foo	2bc680de1e	120472
	jacobs	ca638d5e06	24958
	ucalgary.ca	585cc614	325
	Average		47552

Table 1: Brute-force salt search for a specific five-byte run.

These five bytes correspond on the Intel x86 to a relative jump to the address 12345678_{16}, assuming the jump instruction starts at address zero.

The search was run on an AMD AthlonXP 2600+ with 1 GiB RAM, running Linux 2.6. We tested five different keys with one- to five-byte salts, sequentially searching through the possible salt values.[2] Table 1 shows the results for two cryptographic hash functions, MD5 and SHA-1. For example, the salt "07e9717a09," when concatenated onto the key "aycock," yields the SHA-1 hash value

ef 6d f4 ed 3b a1 ba 66 27 fe e9 74 56 34 12 a2 d0 4f 48 91

Numbering the hash value's bytes starting at zero, our target run is present with $lb = 10$ and $ub = 14$.

Another question is whether or not every possible run can be produced. Using the key "aycock," we were able to produce all possible three-byte runs with three bytes of salt, but could only produce 6% of four-byte runs with a three-byte salt. With a four-byte salt, we were able to generate four-byte runs which covered between 99.999–100% of the possible combinations – this was checked with five different keys and three different cryptographic hash functions. (Our test system did not have sufficient memory to record coverage data for five-byte runs.) The four-byte run data are shown in Table 2.

These results tend to confirm our probability estimate from Section 2: b-bit runs need $b - 1$ bit of salt. Four-byte runs are of particular interest for portability reasons, because RISC instruction sets typically use instructions that are four bytes long; this means that at least one RISC instruction can be generated using our technique.

[2]For implementation reasons, we iterated over salt values with their bytes reversed, and didn't permit zero bytes in the salts.

Algorithm	*Key*	*Runs Found*	*Runs Not Found*
MD5 (128 bits)	aycock	4294936915	30381
	degraaf	4294937044	30252
	foo	4294936921	30375
	jacobs	4294937188	30108
	ucalgary.ca	4294936946	30350
	Average	4294937003	30293
SHA-1 (160 bits)	aycock	4294966707	589
	degraaf	4294966733	563
	foo	4294966660	636
	jacobs	4294966726	570
	ucalgary.ca	4294966769	527
	Average	4294966719	577
SHA-256 (256 bits)	aycock	4294967296	0
	degraaf	4294967296	0
	foo	4294967296	0
	jacobs	4294967296	0
	ucalgary.ca	4294967296	0
	Average	4294967296	0

Table 2: Generation of possible four-byte runs using a four-byte salt.

4 Countermeasures

An analyst who finds some resistant code has several pieces of information immediately available. The salt, the values of lb and ub, and the key's domain (although not its value) are not hidden. The exact cryptographic hash function used can be assumed to be known to the analyst, too – in fact, resistant code could easily use cryptographic hash functions already present on most machines.

There are two pieces of information denied to an analyst:

1. The key's value. Unless the key has been chosen from a small domain of values, then this information may not be deducible. The result is that an analyst may know that a computer virus using this anti-disassembly technique targets someone or something, but would not be able to uncover specifics.

2. The run. If the run is simply being used to obscure the control flow of the resistant code, then an analyst may be able to hazard an educated guess about the run's content. Other cases would be much more difficult to guess: the run may initialize a decryption key to decrypt a larger block of code; the entire run may be a "red herring" and only contain various NOP instructions.

   Note that even if the run is somehow known to an analyst, the cryptographic hash function cannot be reversed to get the original key. At best, the analyst could perform their own brute-force search to determine a set of possible keys (recall that the hash function is many-to-one).

Whether or not every last detail of the resistant code can be found out is a separate issue from whether or not a computer virus using resistant code can be detected. In fact, there is malware already that can automatically update itself via the Internet (e.g., Hybris [4]), so complete analysis of all malware is already impossible.

Fortunately for anti-virus software, computer viruses using the technique we describe would present a relatively large profile which could be detected with traditional defenses, including signature-based methods and heuristics [13]. Precise detection does not require full understanding.

5 Enter the Botnet

What if the computing power available for a brute-force salt search were increased by five orders of magnitude over the computer we used for our experiments? Few organizations have that much computing power at their fingertips, but a few individuals do. A *botnet* is a network of malware-controlled, “zombie” machines that execute commands typically issued via Internet Relay Chat (IRC) channels [3]. These have been used for sending spam and distributed denial-of-service attacks [3], but they may also be viewed as very large-scale distributed computing frameworks which can be used for malicious purposes.

If a virus writer wants to armor a virus using the anti-disassembly technique described here, especially for long runs with many instructions, an already-existing botnet may be used for salt computation. A naïve salt computation on a botnet would involve partitioning the salt search space between machines, and the key and desired run would be available to each machine. This would not be a bad thing from an analyst’s point of view, because locating any machine in the botnet would reveal all the information needed for analysis.

A more sophisticated botnet search would do three things:

1. Obscure the key. A new key, key', could be used, where key' is the cryptographic hash of the original key. The deployed resistant code would obviously need to use key' too.

2. Supply disinformation. The virus writer may choose lb and ub to be larger than necessary, to mislead an analyst. Unneeded bytes in the run could be NOP instructions, or random bytes if the code is unreachable.

3. Hide the discovery of the desired run. Instead of looking for the exact run, the botnet could simply be used to narrow the search space. A weak checksum could be computed for *all* sequences of the desired length in the hash function’s output, and the associated salts forwarded to the virus writer for verification if some criterion is met. For example, the discovery of our five-byte run in Section 3 could be obliquely noted by watching for five-byte sequences whose sum is 505.

This leaves open two countermeasures to an analyst. First, record the key' value in an observed botnet in case the salt is collected later, after the virus writer computes and deploys it – this would reveal the run, but not the original key. Second, the analyst could subvert the botnet, and flood the virus writer with false matches to verify. The latter countermeasure could itself be countered quickly by the virus writer, however, by verifying the weak checksum or filtering out duplicate submissions; in any case, verification is a cheap operation for the virus writer.

6 Related Work and Conclusion

There are few examples of strong cryptographic methods being used for computer viruses – this is probably a good thing. Young and Yung discuss cryptoviruses, which use strong cryptography in a virus' payload for extortion purposes [15]. Riordan and Schneier mentions the possibility of targeting computer viruses [10], as does Filiol [5]. The latter work is most related to ours: it uses environmental key generation to decrypt viral code which is strongly-encrypted. In our case, however, the code run never exists in an encrypted form; it is simply an interpretation of a cryptographic hash function's output. Our technique is stronger in the sense that the ciphertext is not available for analysis.

The dearth of strong cryptography in computer viruses is unlikely to last forever, and preparing for such threats is a prudent precaution. In this particular case of anti-disassembly, traditional defenses will still hold in terms of detection, but full analysis of a computer virus may be a luxury of the past. For more sophisticated virus writers employing botnets to find salt values, and possibly longer runs, proactive intelligence gathering is the best bet.

7 Acknowledgments

The second and third authors' research is supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[3] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *USENIX SRUTI Workshop*, 2005.

[4] F-Secure. F-Secure virus descriptions: Hybris, 2001. `http://www.f-secure.com/v-descs/hybris.shtml`.

[5] E. Filiol. Strong cryptography armoured computer viruses forbidding code analysis: The Bradley virus. In *Proceedings of the 14th Annual EICAR Conference*, pages 216–227, 2005.

[6] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 304–314, 2002.

[7] Krakowicz. Krakowicz's kracking korner: The basics of kracking II, c. 1983. `http://www.skepticfiles.org/cowtext/100/krckwczt.htm`.

[8] R. W. Lo, K. N. Levitt, and R. A. Olsson. MCF: a malicious code filter. *Computers & Security*, 14:541–566, 1995.

[9] H. Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 122–126, 1987.

[10] J. Riordan and B. Schneier. Environmental key generation towards clueless agents. In *Mobile Agents and Security (LNCS 1419)*, pages 15–24, 1998.

[11] R. Rivest. The MD5 message-digest algorithm. RFC 1321, April 1992.

[12] B. Schneier. *Applied Cryptography*. Wiley, 2nd edition, 1996.

[13] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[14] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. `http://eprint.iacr.org/`.

[15] A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *IEEE Symposium on Security and Privacy*, pages 129–141, 1996.