

Malicious cryptography, part two

Frederic Raynal 2006-05-16

This two-part article series looks at how cryptography is a double-edged sword: it is used to make us safer, but it is also being used for malicious purposes within sophisticated viruses. Part two continues the discussion of armored viruses and then looks at a Bradley worm - a worm that uses cryptography in such a way that it cannot be analyzed. Then it is shown how Skype can be used for malicious purposes, with a crypto-virus that is very difficult to detect.

Introduction

In [part one of this article series](#), the concepts behind cryptovirology were discussed. Two examples of malicious cryptography were used, involving weaknesses in the SuckIt rootkit and the potential for someone to design an effective SSH worm. The concept of armored viruses were also introduced.

Now in part two, a continued discussion of armored viruses (using polymorphism and metamorphism) will be followed by the concept of a Bradley worm - a worm that uses cryptography so that it cannot be analyzed. The reader will then look at Skype (now owned by eBay) as an example of an application with embedded cryptography and a closed protocol that can be manipulated by an attacker for malicious purposes, making a virus using this approach very difficult for administrators and anti-virus companies to detect.

A brief review: armored viruses

Part one introduced armored viruses, so let's just quickly review that material. We saw that time is a very critical factor for both attack and defense of a virus. Similar techniques used to protect the intellectual property in software is also being used in malware, and for almost the same purposes. A virus employing techniques to avoid or delay the analysis becomes what is called an armored virus. The first public armored virus fulfilling this goal was called Whale and first spread sometime in September 1990. It combined several techniques:

- Polymorphism: both the binary and the process were ciphered (there were 30 hardcoded versions).
- Stealth: several interruptions, including debugging ones, are hooked by Whale, and it also hides in high memory before decreasing the max limit of memory known by DOS, which was prominent at the time.
- Armoring: the code changes depended on the architecture (8088 or 8086), had intense use of obfuscation (useless code, identical conditions, redundant code, and so on) and had what is known as anti-debug (if a debugger is detected, the keyboard is blocked and Whale kills itself).

If these techniques are common nowadays, almost sixteen years later, one can imagine what

happens then when such a piece of code reaches the labs of the anti-virus companies.

Now let's jump into the new material and take a closer look at armored viruses.

Shape Shifting (polymorphpism and metamorphism)

I particularly enjoy the definition given by Fred Cohen of a virus:

A virus is a succession of instructions which, once interpreted in the right environment, changes others successions of instructions so that a new copy (optionally different) of itself is created in this environment.

What is particularly interesting here is that the definition already assumes a single virus can have multiple representations. Hence, Cohen also defines two other important notions: a *viral set* and the *evolution*:

- A virus is not defined by a single representation but by the set of all its semantically equivalent representations.
- The evolution of a virus is the action of one representation changing to another one in the same viral set

Therefore, well-known polymorphism and metamorphism approaches are simply ways for the virus to copy itself differently. But let's look closer at these two techniques.

Polymorphism is a technique used to encrypt the body of a virus, and to create a different deciphering engine and key each time the virus copies itself (see Figure 1, below). As a very rudimentary example, imagine the cipherring instruction is successively an `ADD`, `XOR`, ... and that the key changes at each execution. More advanced (and efficient) techniques include:

- Out-of-order decoder generation: change the order of the nodes in the graph of instructions (compute the length, retrieve `esp`, deciphering instruction, the loop, ...)
- Pseudo-random index decryption: instead of deciphering the data linearly, the index changes randomly
- Multiple code paths: write the same thing in different ways (`xor %eax, %eax` and `movl $0,%eax`)
- Junk code: insert useless instructions in between useful ones
- Register randomization: registers are not pre-assigned to given instructions, but chosen differently for each new generated code

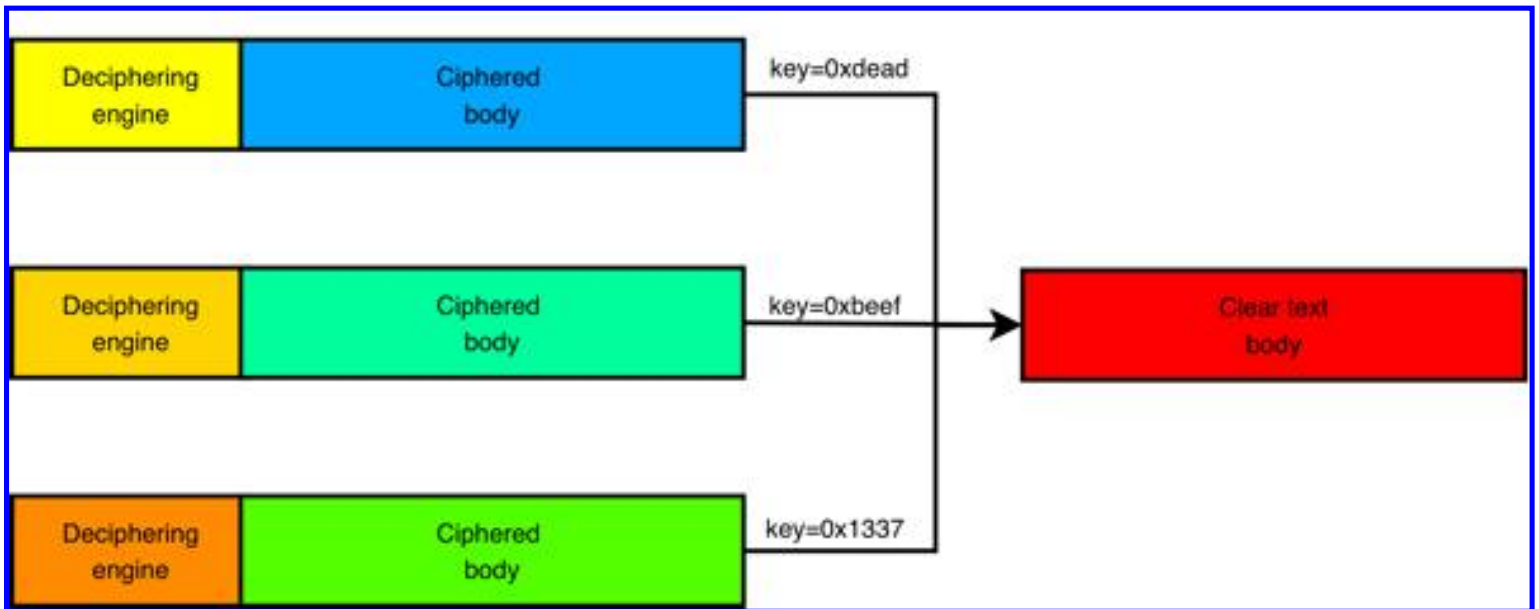


Figure 1. Different polymorphized copies of the same clear-text virus.

The main weakness of polymorphism is that the deciphered code is always the same, which makes runtime detection much easier. In order to avoid that, metamorphism goes one step further.

Metamorphism is a technique to change the full code of a program each time it copies itself (see Figure 2, below). Polymorphism can be seen as metamorphism specialized to a deciphering routine. Very poor metamorphism can be performed by adding junk code between instructions, permuting use registers and so on. More advanced (and of course, efficient) techniques include:

- Code permutation: reorder subroutines, blocks in subroutines, ...
- Execution flow modification: insertion of `jmp` and `call`, tests, ...
- Code integration: code is inserted into another piece of code, and relocation and data references are updated accordingly (an example of this is the virus ZMist)

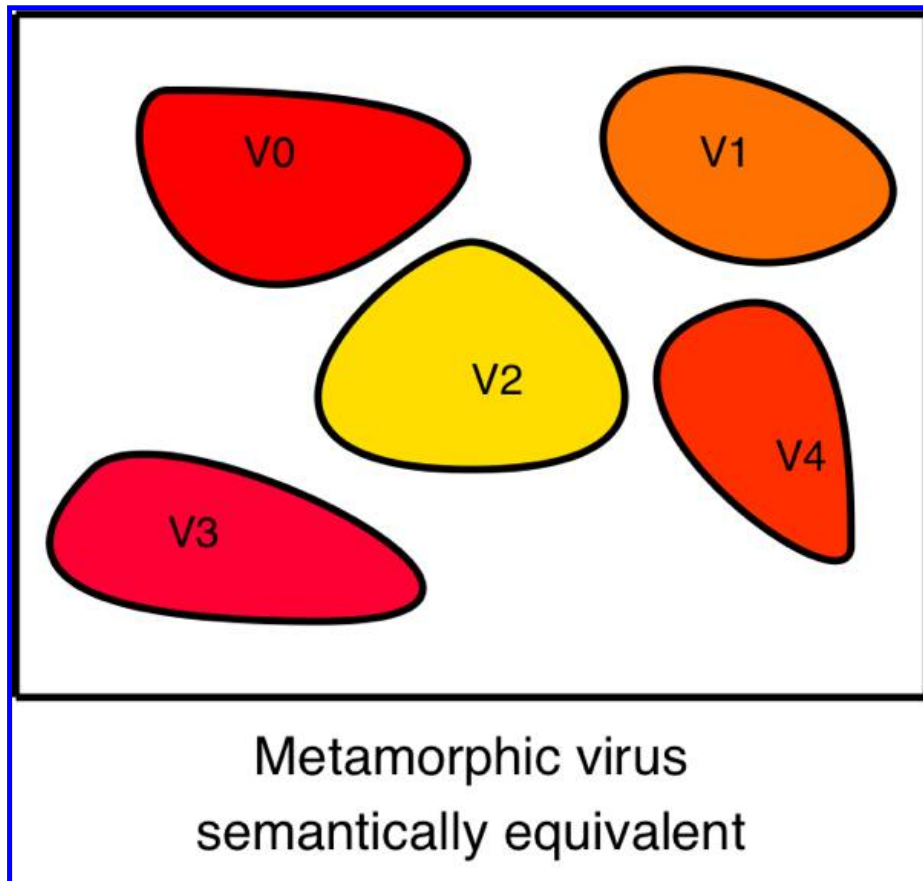


Figure 2. Copies of the same virus with different syntax.

For those who want to have a deeper look at metamorphism, "Metamorphism in practice" [ref4] is a must-read article. It describes a metamorphic engine which is 90% of the code of the virus:

- Viral code is disassembled into an intermediate form
- Redundant and useless instructions are removed
- Transformations (permutations, registers randomization, and so on) are performed on the clean code
- Redundant and useless instructions are inserted
- Code is reassembled and added to the infected files

Here are some final words about polymorphism. It is not only used by malware, but also in shellcode. Designing a good polymorphic engine is, fortunately, not that easy. For malware, it must ensure there is a good randomness between different copies of the malware, otherwise the "fixed points" can be used as signatures. This is beyond the capability of most virus writers.

For shellcode, ciphering instructions can produce forbidden values (such as an unexpected 0x00 in a middle of a `strcpy()`). Moreover, the multiple `NOP` that are usually placed right before the shellcode are not part of the ciphered instructions, and must then also be mutated (note that many IDS do not detect anything if you change a `NOP` for another instruction). But the main cryptographic weakness is that in both cases, the key is present in the code, which makes the analysis easy for both humans and emulators.

So the next question we should ask is: can we build malware without a key or its decoder? You might wonder why we would do that. It is just a matter of tactics. If an attacker removes the key, and the key space is large enough, the right key can not be retrieved. If they remove the deciphering loop, the key is useless as long as the cryptanalyst can not guess what cipher has been used. Hence, the piece of code may evade detection (emulators, IDS, AV) and it can not be analyzed (or the analysis is delayed). But there is may be more options as well.

I lost my keys!

All ciphering algorithms are not equivalent. Some are easy to cryptanalyze, like a XOR ciphering with a short key: an exhaustive attack can be performed. However, if the malware author uses a well-tested algorithm, there is unfortunately no way to retrieve the key. Let us call the first situation outlined as *weak* crypto, and the second one *strong* crypto.

If the malware author removes the key, he must find a way to provide it to his piece of code. Thus, the deciphering loop must be preceded by a key computation step.

As we have seen, when we are using weak crypto, the cipher can be broken using an exhaustive attack. Hence, the key recovery step is this attack: it retrieves the right key by exploring the search space. This kind of technique is called *Random Decryption Algorithm* (RDA). There are 2 kinds of RDA:

- Deterministic RDA: the computation time is always the same (such as was found in the virus W32/Crypto)
- Non-deterministic RDA: the computation time can not be guessed (such as with RDA Fighter or W32/IHSix)

Let us go back to the tactical considerations. Weak cryptography is used to be broken, but it still affords the malware enough time to propagate. It also gives a hard time to the emulator if the loop lasts too long. The next question is therefore: can a malware author do it with strong crypto?

Bradley

Bradley [ref 5] is an un-analyzable virus. The general structure is composed of four parts (see Figure 3, where *EVP* stands for *Environmental Viral Payload*):

- A deciphering function *D* gathers the information needed to compute the key and decipher the corresponding code.
- Encrypted code EVP_1 (key k_1) contains all the anti-anti-viral mechanisms.
- Encrypted code EVP_2 (key k_2) is in charge of the infection and polymorphism/metamorphism mechanisms.
- Encrypted code EVP_3 (key k_3) contains all the optional payloads.

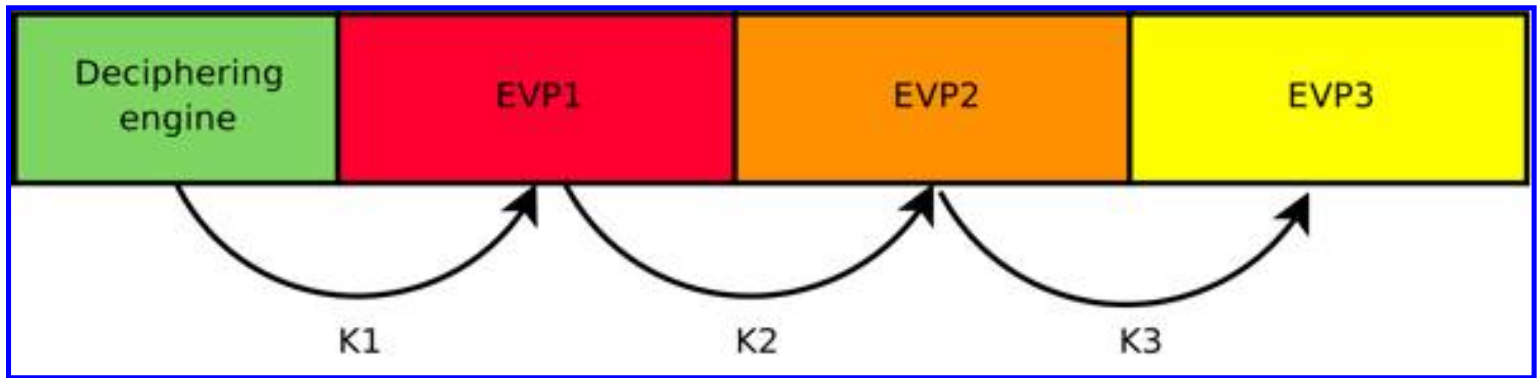


Figure 3. Structure of the virus Bradley.

Most of the crypto used by Bradley is based on the notion of *environmental keys*, as defined by Riordan and Schneier in 1998. It starts with the notion of key exposure: a mobile agent evolving in a hostile environment can not embed keys because if it is captured, key recovery is immediate and so is its analysis.

Hence, their idea was to build keys "on demand," based on the environment of the mobile agent. Let n be an integer corresponding to an environmental observation, H a hash function, m the hash of the observation n (which will act as an activation value) and k a key. We can define several protocols for key computation. Let us consider two examples:

1. if $H(n) == m$, then let $k = n$ (problem: the key transits in clear text).
2. if $(H(H(n))) == m$, then let $k = H(n)$. Here the security of k is equal to the security of H , but replays are possible.

These two examples are not very cryptographically resistant, but are provided to give the reader the general idea.

What about the opportunity for environmental observation? There are so many possibilities: time, hash value of a given web page, hash of a RR in a DNS answer, the content of a file on the targets, information contained in an e-mail, weather temperature or stock value, and so on.

So, in Bradley, the deciphering function D needs some information under the control of the attacker to correctly compute the activation value, and then the keys.

The complex part with such malware is to define a clean and efficient key management. So, D gathers the needed information, including the one under the control of the attacker (let it be called a). It computes $H(H(n+a)+e_1)$, where e_1 denotes the first 512 bits of the encrypted code EVP_1 , and if it is equal to the activation value m , then the key needed to decipher EVP_1 is $H(n+a)$. Otherwise, D disinfects the system from the whole viral code. Next, the nested key $k_2 = H(c_1+k_1)$ is computed (c_1 are the 512 last bits of the clear text code VP_1). And so on for EVP_3 and k_3).

For replication, the strategy is to change everything, including the activation value. This can be done by updating the environmental information n , and thus re-compute a new m' and k_1' .

And since the encrypted code EVP_1 changes, so do the nested keys, and the ciphered code.

All mathematical details are explained in, "Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the Bradley virus". [ref 5] Interested readers should definitely have a look at the reference. What is particularly interesting is that it proves that the analysis of a code protected by the environmental key generation protocol defined therein is a problem which has an exponential complexity.

From an operational point of view, information leakage is restricted to e_1 and to the scan for the given information a , but one can not retrieve it because of the use of the hash function. So, even if this code is detected and put in a lab for study, it wont be analyzed unless this information is retrieved.

Of course, this property itself is really interesting. But there is more an attacker can do with Bradley. If he combines both environmental keys and polymorphism, he effectively get surgical strikes. Assuming the environmental keys depend also on the target:

- An analyst has no possibility of identifying who is the target.
- The attacker gains good control on the spreading of the malware:
 - If the target is a person, one can use his email address, or even better, his public key (GPG or SSH, for exampe - after all, public keys are designed to identify a person)
 - If the target is a "group," one needs to find information specific to this group, such as a domain name for a company, or TLD for a country, and so on.

Analysts needs to understand the impact of a Bradley virus and the implications of how effective it could be used my a attacker to commit a crime.

A matter of stealth

An effecitive way for malware not to be eliminated is for it first not to be detected. Let us now see how malware authors try to improve the stealth of their code or their attacks using cryptography.

No deciphering loop?

Let us go back to the structural consideration we had previously discussed, on polymorphism engines. We have already seen that it is possible to avoid embedding a key in the malware.

So, now we have a key and encrypted data that needs to be decrypted. Thus, we need to find a deciphering loop (and moreover it must be the exact inverse function of the ciphering one). Malware is like a parasite, so malware authors try to let them be even more parasitic: in this example, we will use the crypto provided on the infected system (such as CryptoAPI in Windows, or OpenSSL in Unix). Let us now use the CryptoAPI as an example:



```

int main(int argc, char *argv[])
{
    HCRYPTPROV hCryptProv;
    HCRYPTHASH hCryptHash;
    HCRYPTKEY hCryptKey;
    BYTE szPassword[] = "...";
    DWORD i, dwLength = strlen(szPassword);
    BYTE pbData[] = "...";

    CryptAcquireContext(&hCryptProv, NULL, NULL, PROV_RSA_FULL, 0);
    CryptCreateHash(hCryptProv, CALG_MD5, 0, 0, &hCryptHash);
    CryptHashData(hCryptHash, szPassword, dwLength, 0);
    CryptDeriveKey(hCryptProv, CALG_RC4, hCryptHash,
                  CRYPT_EXPORTABLE, &hCryptKey);
    CryptEncrypt(hCryptKey, 0, TRUE, 0, pbData, &dwLength, dwLength);
}

```

Deciphering is done by replacing `CryptEncrypt()` by `CryptDecrypt()`, and the algorithm used is changed by modifying a single constant.

Building shellcode using the CryptoAPI is exactly the same as what is usually done when writing a shellcode for Windows. It starts by looking up `kernel32.dll`, and then `GetProcAddress()` and `LoadLibrary()`. Now, one has to load `advapi32.dll` and find the previously mentioned functions, then call them to decipher the code before running it.

This overall approach can be interesting for multistage shellcode which will then protect the information sent to the target host. For malware, using big and complex external libraries can make the work of emulators much more complicated. The main problem with such a trick is that the sequence of functions is easily recognizable.

Embedded crypto abused

Skype can be considered by some researchers to be a naturally armored, human-propagating virus. This statement is meant to be humorous, of course, but we will see as a case study what this actually implies and what can be done with all the available features of an application like Skype that has embedded crypto. [\[ref 6\]](#)

The Skype binary is protected with several layers of ciphering, and many integrity checks are spread all around the code. From a network perspective, Skype is able to defeat any firewall by tunneling its traffic. It attempts to connect to a login server and other specific servers through port 80 or 443. But if these ports are not open, it also tries other ports with TCP and UDP. Moreover, the protocol is fully closed: no documentation is available. But above all, users click and install this application it confidently.

Authentication is based on a central server owned by Skype (and its parent company, eBay). The binary embeds 13 public keys related to Skype and the login server. When a client logs in:

- A 1024 bits RSA key (p, s) is generated.
- A session key k is generated.
- The user provides his password.

Then, a computation is made, and authentication data is sent to Skype's login server:

$$\text{RSA}_{\text{skype}}(k \parallel \text{AES256}_k(p \parallel \text{MD5}(\langle \text{login} \rangle \parallel "\backslash\text{n}\text{skyper}\backslash\text{n}" \parallel \langle \text{password} \rangle))))$$

So, RSA is used with one of Skype's public keys. It is used to cipher the session key k. This key k is used to cipher the new public key p of the user with AES256. Last but not least, an MD5 hash of the authentication information is sent, which is the login and the password. Hence, one just needs to get the MD5($\langle \text{login} \rangle \parallel "\backslash\text{n}\text{skyper}\backslash\text{n}" \parallel \langle \text{password} \rangle$) to log in as a given user. All other information is either public (Skype's key) or provided on demand.

From a cryptographic point of view, the main flaw is that replay is possible. Thus, if someone (other than Skype) can talk Skype and capture the traffic, he can re-use the authentication information to impersonate a user.

Now let's see what could be done maliciously with Skype by an attacker.

From the inside, many types of crypto are used, but there is also compression, and all of these are with an undocumented protocol: no one will be able to detect malicious traffic, which is perfect for a malware's stealth. However, compared to SSH or SuckIt as we have previously seen, there is no way to execute arbitrary code on a client: the attacker needs an application level flaw.

From the outside, the connection between clients looks to be direct, even if it is proxied by other nodes or supernodes. This, unfortunately, would be excellent for the accuracy of a worm in such an environment. And finally, the P2P infrastructure itself is very redundant and dynamic, which provides a virtual playground for the survivability of malicious malware. Hence, once a flaw is found in the Skype application (or in an external library used by Skype), a worm would have a very, very good network with which to grow.

Let's imagine a worm for this example. It could exploit a remote flaw in a single UDP packet or a few TCP ones (this is not fiction or a dream ... see the Black Hat Europe presentation, "Silver Needle in the Skype". [ref 6]. The worm could bypass firewalls to reach LANs, and connect to/from the LAN to anywhere on the Internet. Of course, it could also use the "secure channel," that is, the closed-but-encrypted protocol to avoid detection by IDS/IPS. And thanks to the P2P infrastructure and some of its inherent features, accuracy would be around 100%:

- If the worm is on a client, it could propagate using the "search for buddy" algorithm and check for their presence on the network.
- If the worm is on a supernode, the worm could attack all its clients.

With something like five million users using Skype at any one time, one can imagine what such a worm could do. It could be catastrophic. The payload could be anything, but just as an example it could change Skype's public keys and disconnect the user... who will not be able to reconnect.

This is the central idea behind creating a *Skype Private Network*. If someone cleans the binary (in other words, removes the ciphering layers, integrity checks, and so on), and then changes the public keys embedded in the binary, but also some hard coded IPs (login servers, supernodes), and if he provides his own login servers and supernodes, the attacker would have a private network under his control. This addresses one of the main criticisms of Skype. Since Skype is its own certification entity, it can do anything it wants. Hopefully the security community can open the eyes of Skype developers to this pending problem.

Final words

There are many other ways where crypto could be use to improve the efficiency of a malicious attack. For instance, consider the survivability of malware. A malicious coder could try to make some kind of immortal malware, something as pervasive as Explorer running on Windows. But the attacker might also make the malware become more valuable or integral to the user when it is alive and not killed/removed - such that even if it is detected, the user would not want to delete it because the death of the malware will result in a loss of critical data that is simply too valuable for the user.

Examples used in this article series have been taken from different domains, from the exploitation of human beings (using SSH) to strong crypto that is badly used (the SuckIT rootkit, or a malicious use of Skype). In reality, we only have focused on three properties in this article: accuracy, time and stealth.

We have also seen that all this is a matter of perspective. Polymorphism can be used in a defensive way to protect the analysis of a code, in a neutral way to avoid detection, and in an offensive way for surgical strikes. This is all the same for crypto. We must understand the approaches used by sophisticated attackers if we are to defend against them.

Author acknowledgments

Special thanks to all of those who helped me to study this topic and write this article, providing ideas, comments, diet coke and much more.

References

[ref 4] [Metamorphism in practice](#)

The Mental Driller, 29A, vol. 6

[ref 5] Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the Bradley virus

Éric Filiol, Proceedings of the 14th EICAR Conference, 2005

[ref 6] [Silver Needle in the Skype](#)

P. Biondi, F. Desclaux, Black Hat Amsterdam, 2006

Disclaimer

The nature of cryptography has become one where it can be used for malicious purposes just as it is used to make systems and networks more secure. SecurityFocus, and its parent company Symantec, do not necessarily endorse the approaches discussed in this article. The editorial team abides by the ethical mantra of Do No Evil, and we believe that cryptographic approaches must be discussed from all angles if they are to truly keep us safe.

[Privacy Statement](#)

Copyright 2006, SecurityFocus