DYNAMIC DECRYPTION PROCEDURES IN MALWARE

Omar Alejandro Herrera Reyna
oherrera@prodigy.net.mx
September 23rd 2006

Abstract

Reverse engineering is regarded as one of the most trusted techniques to gather information about malware. It has been considered by many as a fail-safe method to completely understand the characteristics and operation of malware in a reasonable amount of time (for most cases). In recent years, research on cryptographically armored malware has produced techniques that can complicate reverse engineering to a degree where the information provided is not useful anymore (not within reasonable time at least). This paper shows that the threat of dynamic decryption procedures within malware is real and that their use is feasible in several types of attacks.

Keywords

Malicious software, malware, dynamic decryption, reverse engineering, targeted attacks, malware analysis, cryptographically armored malware.

1. INTRODUCTION

In recent years malware researchers have been discussing the limits of reverse engineering as a technique to effectively discover how malware works in any situation. Several researchers argue that reverse engineering is the most effective tool to understand malicious software while other activities such as running the malware in a controlled environment are time consuming and their effectiveness is dubious.

Riordan and Schneier devised a technique where the content of mobile agents could be protected from analysis while they travel through insecure networks; they also devised the malicious application of this technique when applied by directed viruses [1]. The agents and viruses in this context would carry sensitive parts (information or code) encrypted but they wouldn't carry the key. Instead, the key would be constructed by taking information from environmental variables.

A similar technique can be applied to other types of malware (e.g. trojan horses) in order to hide parts of their code and preventing analysis in a reasonable amount of time. Research on the uses of cryptography in malware has been going on for a number of years. Young and Yung have described several methods of extortion using cryptography within malware [2] and Filiol created a hypothetical cryptographically armored virus, the Bradley virus, that uses environmental key generation [3]. Also, Hirt and Aycock mention techniques for anonymous communication between malware programs, some of which involve the use of encryption [4].

Based on the idea of applying environmental key generation [1] to create dynamic decryption procedures (DDP) within malware, the following sections will describe and analyze different implementations and how practical they are for malware authors.

2. ELEMENTS AND STRUCTURE OF A DDP

Definitions and theory

An environmental key is formed from data gathered from different sources available in the environment. This environment, in the context of malicious software, is the place

where it is executed (i.e. the operating system, applications, and hardware devices with which it can interact). Before digging into the details of the structure, a few definitions are necessary.

Let *S* be a source of data potentially available to the malware through interaction with its environment (we say potentially since not all environments are equal, and not all data sources might be available in all environments). Each data source will return a number of bits of data when queried. So, let us define *M* as these bits of data (which we will also call the message) and define *Q(S)* as the query function applied on source *S*. Hence, to query data from source S_1 to obtain message M_1:

$$Q(S_1) = M_1$$

Any environment possesses a finite number of sources of data. So let's define the Environment *E* as a set of *N* data sources as follows:

$$E = \{S_1, S_2, ..., S_N\}$$

Malware do not need to use all sources of data available to generate the keys
So we only need a subset of sources of data which we define as *K*:

$$K \subseteq E$$

In practice it would be nearly impossible to use all sources of data, so in most real cases *K* would be a proper subset of *E* and the number of elements in *K* is much smaller than those in *E*:

$$K \subset E \text{ and } 0 < |K| << |E|$$

To create an environmental key *EK* in the context of a DDP, we need a constructor function, *CF*, which takes as input all elements of *K*. The result is a bit string of fixed length that the malware can use as a decryption key (*EK*). Therefore:

$$CF(K) = EK$$

Obviously, *CF* will make use function *Q* internally, with each data source that is included in *K*:

$CF_T(K) :=$ A transformation on the message set $\{M | M = Q(S), \forall S \in K\}$ at a specific time *T*.

Note that *CF* is a generalization of all possible functions that can be used to produce *EK* from *K*. Riordan and Schneier provide some examples of constructor functions that make use of one-way hashing functions [1].

We also need a condition to specify when the decryption will be attempted in time. This condition *CND* is a function applied to message M_S (which in turn is queried from a

specific source of data S_S that it is part of the set of data sources[1] in *E*) and returns a binary response: 0 if the condition is not met, and 1 if the condition is met:

$S_S \in E$, where S_S could be part of *K*.

$Q(S_S) = M_S$

$CND(M_S) = 0$ if M_S does not meet the required condition or 1 if it does.

In addition, we need to define the part of the malware that gets encrypted, *P* (the payload), the encrypted version of the payload, *EP*, the function to encrypt the payload *C*, and the function to decrypt the payload, *D*. Both *C* and *D* use a specific cryptographic algorithm (*A*) and take two parameters: the bit string that will be encrypted/decrypted and the corresponding encryption/decryption key.

$C_A(P,Y) = EP$

$D_A(EP,Y) = P$

$D_A(C_A(P,EK),EK) = P$

And so on.

An execution function *EX* that is able to execute the code of the decrypted payload is necessary as well.

Finally, the malware needs to know when the decryption has been successful (meaning that the required environment conditions have been satisfied), in order to execute the decrypted payload. For this we will define function *V* that returns a verification code. *VC* will be the verification code corresponding to the unencrypted payload *P*:

$V(P) = VC$

$V(D_A(EP,EK)) = V(P) \Rightarrow$ Decryption was successful.

$V(D_A(EP,EK) \neq V(P) \Rightarrow$ Decryption was not successful.

Let us now summarize what we actually need to create a DDP, so that it can be used within a malware program:

- A set of data sources for the encryption/decryption key: *K*
- A function to query data sources to get messages: *Q*
- A function to generate a key from a set of data sources: *CF*
- A data source and a condition function to test when the decryption must be attempted: S_S and *CND*
- Encryption and decryption functions for the chosen cryptographic algorithm, *A*: C_A and D_A
- A verification function that will let us know if the decryption attempt was successful or not: *V*
- A verification code for that corresponds to the unencrypted payload *P*: $VC = V(P)$

[1] Strictly speaking, this should be a set of data sources, like *K*, but we will assume it is a single data source for simplicity.

- The encrypted version of payload *P* with a secret key *Y*: $EP = C_A(P,Y)$
- Memory storage for the calculated environmental key, the decrypted payload, and the message queried from the conditional data source: *EK*, *TMP* and M_S
- An execution function to transfer control to the successfully decrypted payload: $EX(TMP)$

We can now define the general structure a DDP using pseudo-code[2]:

```
void DDP (){
   constant bitArray VC = {...};
   constant bitArray EP = {...};
   constant sourceArray K = {...};
   constant dataSource Ss = {...};
   variable dataMessage Ms;
   variable bitArray TMP;
   variable key EK;
   while(TRUE){
      Ms=Q(Ss);
      if(CND(Ms)==1){
         EK=CF(K);
         TMP=(Da(EP,EK);
         if(V(TMP)==VC){
            EX(TMP);
         }
      }
   }
}
```

That's it. The malware author would just need to define *CF*, *CND* and *K* (by selecting a number of data sources available from the environment), and select a data source S_S for the condition to attempt decryption and calculate the constant values *VC* and *EP*.

Implementation considerations in practice

There is a wide range of data sources that the attacker can use as inputs in order to generate the decryption key. The following types of sources of data are just some examples:

- Hard disk activity – any statistics or events (e.g. reading, writing, volume of data transfers in a fixed time frame, number of hard disk operations in a fixed time frame, etcetera)
- Network activity – any statistics or events (e.g. receiving data, sending data, number of packets per second in a fixed time frame, number of established connections at a specific time, transfer speed above a fixed threshold)
- System uptime
- System date and time data
- Processes – any statistics or events (e.g. number of processes running at a specific time, if specific processes are running/not running, average running

[2] For readability, Subscript letters were replaced with lowercase letters when using a mono-spaced font in this example.

time of processes, running speed of a specific test function above a certain threshold).
- Input/Output devices – their existence, brand/driver or data obtained from any of those devices (e.g. keyboard, mouse, printer)
- Hardware identification data (e.g. MAC address of network interfaces, CPU ID)

For the decryption condition (*CND*), a useful data source could be the time. For example, the condition could be built in such a way that the decryption would be attempted each N seconds or minutes. Since encryption/and decryption are CPU intensive operations, trying decryption operations one after another would be too obvious (turning on a laptop's fans for example), so it makes sense to slow down a bit.

There are a wide range of constructor functions to create the key, some are simple and some are more complex. The complexity doesn't add any security to the DDP; their speed does (a little bit). This will be discussed in more detail in the next section.

For the cryptographic algorithms, a symmetric algorithm (e.g. AES, DES, 3DES, Blowfish, RC4) would be used in most cases. The use of certain public key algorithms that are able to encrypt such as RSA is also possible, but some restrictions need to be taken into account. E.g. the length of the payload in bits must be less than the key length if all decryption is expected to be done in a single call to the RSA function; otherwise the payload needs to be fragmented and the RSA function would need to be called for each encrypted fragment [5]. Combinations are also possible.

In the case of the verification code and verification function, one-way hash functions such as MD5 or SHA-1 are well suited for this task [6], since they allow the detection of a successfully decrypted payload without disclosing any useful information to someone analyzing the malware. The verification code would then be a hash produced by these algorithms.

Finally, there are a few options for the execution function. It should be noted that at least the decrypted payload must reside in volatile memory at some point. It is possible to use shellcode as the payload, in which case a simple jump into the code would suffice. The advantage of this approach is that the decrypted payload is not intentionally written to the disk. Even with the possibility of a memory swap taking place, if the shellcode is quick enough and the decrypted payload is overwritten after it has been executed; the chances of obtaining the decrypted payload from disk are low.

However, the shellcode needs to be aware of some limitations existing in different operating systems. For example, in Microsoft Windows, the shellcode will first need to locate the addresses of any operating system functions that it intends to use (e.g. network, filesystem). It could use the import library table of the hosting program (where the DDP lies), but it is unlikely that all functions required by the malware will exist there[3]. There are different techniques to achieve this, some of which are explained in [7]. Pseudo-code of a DDP that uses shellcode in the payload and a hashing function as the constructor function for the key is illustrated in [8].

[3] Adding all those functions manually to the hosting program library table will leak information to anyone analyzing the malware.

Another, less stealth but easier to implement approach would be to simply include a whole executable file as the payload. The execution function would need decrypt the payload and, in some cases, write it to the disk before executing it. There are some drawbacks with this approach: first, that the decrypted payload is written to disk (increasing the probability of recovery by using digital forensic techniques) and second, that the hosting program (with the DDP) needs to have write and execution privileges on the directory where it chooses to deploy the payload.

In Microsoft Windows, there is no API function to execute a PE program from memory; all programs need to be stored in disk. However there are some techniques to get around this problem that resemble a hybrid of the two approaches mentioned above. Such a technique is described by Kuster [9].

3. REQUIREMENTS FOR A ROBUST DDP

Just because a malware author includes a DDP within its code it doesn't mean that its implementation is robust, from a confidentiality point of view. In this section we will analyze different elements of a DDP that were introduced in the previous section and highlight the requirements for a robust DDP.

Information available to a researcher analyzing a DDP

First of all, let's discuss a hypothetical scenario where a researcher is analyzing a piece of malware implementing a DDP that he was able to capture. If she/he was able to somehow capture the decrypted payload then the situation would become like any other malware analysis case. However it is important to note that this is only possible if the DDP was able to decrypt the payload *EP* at least once, before the analysis took place (i.e. if the decryption conditions are still not met, then this scenario is impossible).

So, let us assume a scenario where the researcher has no access to *P* (yet). What other information could be obtained from the code? Basically, each element that appears in the DDP structure can be recovered. By definition, an encrypted text (the payload in this case) is one that has been disguised by a process called encryption, so that its substance remains hidden [6]. Because the meaning of any encrypted payload is hidden and cannot be used, it its obvious that the only part within a DDP that can be encrypted is the payload itself (producing *EP*); all other elements in the structure are needed for the DDP to work (to be able to decrypt the payload when some predefined conditions are met).

Therefore, all that can be done with any of the elements in the DDP structure other than the *EP* is to obscure their meaning in some way (this includes additional the code within the DDP to link the elements, such as conditional loops and function skeletons). But even if obscured with polymorphic or metamorphic procedures (which might pose significant difficulties for automated detection), Ferrie and Ször suggest that careful manual analysis would still provide an understanding of the code [10, 11]. Thus, we will assume that such techniques can always be circumvented with a proper analysis by an experienced researcher, within a reasonable time.

After this analysis, we can conclude that a researcher in the situation previously mentioned will always be able to identify all elements in the basic structure of a DDP, and have access to their content, with the exception of the encrypted payload (*EP*). The

following table summarizes the elements that can be accessed by the researcher and the malware author in the aforementioned scenario:

Element	Elements accessible to:	
	Malware researcher	Malware author
Code to link other elements within the DDP (e.g. conditional loops and function structures, including the variables *TMP* and *EK*)	●	●
Verification code: *VC*	●	●
The verification function for the decrypted payload: *V*	●	●
The constructor function for de environmental key: *CF*	●	●
The encryption function for the selected cryptographic algorithm: C_A	●[4]	●
The decryption function for the selected cryptographic algorithm: D_A	●	●
The Query function to obtain data (messages) from a data source: *Q*	●	●
The set of the selected data sources to generate the environmental key: *K*	●	●
The data source that defines when de decryption is attempted: S_S	●	●
The condition that triggers a decryption attempt: M_S	●	●
The execution function to transfer control to the payload after a successful decryption: *EX*	●	●
The encrypted payload: *EP*	●	●
The unencrypted payload: *P*		●
The encryption key: *Y* (So that when *EK*=*Y*, $V(D_A(EP,EK)) = V(P)$, meaning that the decryption was successful)		●

This shows that any malware researcher will have access to most of the information with the exception of the unencrypted payload (*P*) and the encryption key (*Y*).

Available options to recover the payload

With the information indicated in the table above at the disposal of the malware researcher, she/he has at least the following options to attempt the recovery of the payload *P*:

[4] Although the encryption function (C_A) is not directly available from the DDP, it shouldn't be impossible to recreate it from the decryption function (D_A), even if the algorithm is not well known. For instance, according to Shepherd, any sequence transformation can be described by finite automata, and if there exist a finite automaton describing an encryptor there must exist a second finite automaton acting as a decryptor that accepts the output sequence of the first automaton (in a corresponding initial state) that produces the original input sequence of the first automaton [12].

1. Cryptanalysis of the decryption function for the specified algorithm (D_A)
2. Black box analysis in a controlled environment (i.e. lab), where the potential malicious software is executed with the hope that the decryption will be successful at some point, so that the payload may be recovered.
3. Brute force attack using all possible messages from each data source in *K*, making use of functions *CF*, D_A and *V* to attempt the decryption.
4. Dictionary attack using reduced (more probable) sets of messages from each data source in *K*, making use of functions *CF*, D_A and *V* to attempt the decryption.

Complicating DDP analysis with a robust design

Malware authors can still complicate the analysis to a degree where it becomes unfeasible to obtain *P* from *EP* in a reasonable amount of time, and all that is required is a careful design.

The first option available to the malware researcher that we mentioned was cryptanalysis. By simply using a well known, strong, encryption algorithm, such as 3DES, AES, IDEA or Blowfish (to name a few) and well tested implementation, the malware author can prevent malware researchers from going this way (unless the researcher has a lot of faith on her/his abilities, available time and resources).

To effectively describe how to defeat the three remaining options available to malware researchers, more analysis and a few additional definitions are required.

We will start by showing that the security of the encrypted payload (*EP*) rests only in 3 places: The decryption function for the specified algorithm (D_A), whose security issues have already been discussed, the set of data sources selected (*K*) and the specific data source (S_S) used to define when the decryption is to be attempted. Filiol reached similar conclusions while analysing the Bradley virus in [3].

Under Kerckhoffs' Assumption, a good cryptographic system should be secure even if the ciphertext and all details of the encryption function are known to the attacker, with the exception of the encryption key (which in turn becomes entirely responsible for the security of the ciphertext) [6,13].

Regarding the key, we know that it is generated by applying the construction function on the set of selected data sources: $EK = CF(K)$. Intuitively, we might think that the constructor function (*CF*) has also some responsibility in determining the security of the generated key (*EK*). However, to be of any use to the malware author, *CF* must produce exactly one output (*EK*) for each combination of messages (*M*), generated by querying each data source (*S*) within our data set (*K*). This is coherent with our definition in section 2:

$CF_T(K)$:= a transformation using the message set $\{M \mid M = Q(S), \forall S \in K\}$ at a specific time *T*.

The other important fact to remember is that the malware researcher has always access to the construction function (*CF*). Therefore, the resulting key *EK* is solely dependant upon the set of data sources (*K*) that is used to create it at any given time.

In other words, Any *CF* function (no matter how complex or simple it is) can only generate as many possible keys as there are message combinations that can be obtained from data sources within *K*. Thus, choosing any particular *CF* does not affect in any way the number of possible keys. Yet, this doesn't mean that *CF* is totally useless. By making the task of changing keys costly in terms of computer cycles, an exhaustive key search can be slowed down [13], and this is where *CF* could be somewhat useful; this will become more obvious when we discuss the last 2 options available to malware researchers (brute force attacks and dictionary attacks).

A keyspace is a range of all possible values that a key can take [6]; in our case, the key (*EK*) if formed from several pieces of data (messages queried from al data sources in the data set *K*) that undergo some kind of transformation (through function *CF*). Hence, the keyspace in the context of a DDP depends on the combinations of the individual keyspace of each data source. Let us then define *KS* (keyspace of the dataset *K*) as the number of all possible key combinations that could be obtained by applying the *CF* function over the set of data sources *K*:

KS := $\left|\left\{EK \middle| EK = CF_j(K), \forall j\right\}\right|$, i.e. the cardinality of the set containing al possible keys (combinations) from *CF*(*K*).

There is an issue with the data source used to determine when to attempt the decryption (S_S). Some data sources are related to other data sources (i.e., they depend on each other to some extent). This means that there is some redundancy in the information obtained from them. If the malware author is not careful, the effective keyspace (keyspace after getting rid of the redundancies), which we will define as *EKS*, can be less than the keyspace *KS*.

It can be shown that using a data source S_S that is also part of the data set in *K* ($S_S \in K$) will reduce the effective key length by a number of combinations. This number of combinations corresponds to the number of possible messages from source S_S that do not satisfy condition tested by the *CND* function. This is obvious since there exists a predefined condition against which M_S (where: $M_S = Q(S_S)$) is always tested before the decryption is attempted.

For example, if the data source S_S is chosen to be the number of seconds in a computer clock and the condition defined by *CND* is that M_S must be even, then the contribution in possible combinations by this data source to the effective key space (*EKS*) is reduced from 60 to 30 (i.e. there are 30 even numbers in the whole range, 0-59, of the 60 possible values). This also illustrates the difference between *KS* and *EKS*. *KS* will take into account all possible combinations (60), while *EKS* takes only into account how many of these combinations are valid within the context of the DDP (30).

Note that both *KS* and *EKS* are directly affected by the number of possible combinations in messages that are queried from the data sources in the set *K*. The fact that these messages are represented efficiently or inefficiently is irrelevant. E.g. we can represent the number of seconds in a clock with no less than 6 bits, which is the most efficient representation, or use instead 2 bytes, one for each ASCII character for the two digits required to represent a second, which is very inefficient. Thus, malware authors and researchers should be aware that lengthy (and inefficient) representations of messages

do not contribute more combinations to the keyspaces *KS* and *EKS* (probably a common mistake when calculating keyspaces)[5].

Now let's return to the remaining three options available to malware investigators to attempt the recovery of the payload *P*.

The second option available for researchers, black box analysis, seems to be a hopeless wait for a miracle, but it might work under some circumstances.

These circumstances depend completely on the data sources selected (*K*), the decryption condition (*CND*) and its corresponding specific data source (S_S). None of our measurements of keyspace (*KS*, *EKS*) are involved here, however. Let *TF* be a time frame (i.e. the interval from the start of the frame to the end of the frame, inclusive), and *p*(*X*) the probability of an event taking place:

Let T_2 and T_1 represent specific times, where $T_1 < T_2$. Then $TF := [T_1, T_2]$

$p_{TF}(X) :=$ The probability of event *X* taking place within the time frame defined by *TF*.

Then, using option 2 is feasible for a researcher if $p_{TF}(V(D_A(EP, CF(K))) = VC) = 1$, for any reasonable *TF*. That is, if the researcher can deduce from *K* a reasonable time frame (e.g. 2 hours, 1 day, etc.) where she/he is absolutely sure that the decryption is going to be successful, then it is reasonable to use black box analysis. The caveat here is that the researcher can only work out *TF* if all *S* (where $S \in K$) depend on time and if the time frame can be translated to the future (otherwise there is no guarantee that the event will take place in the future since it might have already happened).

Two examples might help to clarify why this is so. First, let us assume that our condition (*CND*) will always be true for simplicity. Then let us assume we have 2 DDPs: DDP_1 and DDP_2, which have K_1 and K_2 as their respective data sources. Then, let:

$K_1 = \{S_1\}, S_1 :=$ The current minute indicated by the system clock (0-59).
$K_2 = \{S_2\}, S_2 :=$ The username of the user currently logged in to the system.

There is a single data source in the *K* sets in each case. In the first case we can predict *TF* since we know we can translate this time frame into the future: T_1 could be the current time and $T_2 = T_1$+59 minutes and we are absolutely sure that within one hour, the decryption will be successful (incidentally, we might find that 1 hour is also a reasonable time frame). However, in the second case we have no way to determine neither T_1 *nor* T_2 because the event signalled by a specific message when querying S_2 is independent

[5] Ideally, the keyspace of a cryptographic algorithm should be equal to the key length in bits, that is: all possible values of the key can be used. In practice this is not always the case. For example, assuming an implementation of AES with a 128 bit key length where the key is generated from a lowercase string of 8 characters (assuming an alphabet of 26 characters), we know that the key length is still 2^{128} (~ 3.4×10^{38}) for the algorithm, but that not all of those values will be used, only 26^8 (~2.1×10^{11}) will be used because each of all possible 8 character, lowercase, passwords must map to exactly one 128 bit key value. Something similar happens in a DDP; the effective keyspace depends on the data sources selected and not on the key length of the algorithm selected.

from time (in strict sense, we can't even guarantee that the correct user will ever log in to the system, even if we suspect that the event has not happened yet).

So, we conclude that malware authors can prevent malware researchers from considering option 2 by either adding some time independent data sources into the *K* set and, by making the time frame unreasonably large, or by selecting data sources so that the time frame cannot be translated into the future (e.g. using the year as data source, past years are not going to happen again, or at least they shouldn't show up on the system's clock again).

We still have 2 more options left, so let us analyze the third one: brute force attacks. According to [6,13], a brute force attack is one where the attacker tries every possible key with the help of small amounts of ciphertext and their corresponding plaintext. In the case of DDP we don't have any plaintext, but we have a verification code (*VC*) that is equally useful, because it allows us to determine if the decryption was successful or not[6].

We can of course try every possible combination of key values as produced by *CF*(*K*) with all possible messages of all $S \in K$ (this number of combinations is *KS*). However, there is no reason to do that if we have previously identified invalid combinations (i.e. if *EKS* < *KS).* The largest number of combinations that we would be expected to try in a brute force attack is always defined by *EKS* (whether it is less than or equal to KS).

With an effective keyspace *EKS,* one would expect to find the key after searching half the keyspace [13]. That is, after trying half of the valid combinations (*EKS*/2) there is a probability of 0.5 that the key would be found and that is regarded as a reasonable estimation of the expected number of combinations to try before the key is found (i.e. on average this is the time that will take get the correct key). There are 2 factors that determine the speed of a brute force attack: the number of combinations to try and the speed of each test [6].

Let $XC = (EKS / 2)$ be the number of expected combinations to try.

Let $XT :=$ The time required to conduct each test, from calculating the key *EK* with the constructor function *CF*, to the point where the decrypted payload, $D_A(EP,EK)$, is verified with function *V* and verification code *VC*.

Let $XBR = XC * XT$ Be the expected time required to find the correct key (i.e. break the encryption).

Obviously, the malware author will need to make sure that *XBR* will result in a sufficiently large time. Adding data sources that contribute a considerable amount of combinations is one way to achieve this[7], reducing invalid combinations so that *EKS*=*KS* (or at least is close to *KS*) would also help, and as mentioned previously while discussing the first option, choosing slow functions for *CF* and *V* and a slow (but still robust) cryptographic

[6] By definition, a DDP will always require some method to detect that the decryption was successful (this has been defined so by assuming that jumping into random bits of unknown code, potentially crashing the machine most of the time is useless for malware authors). This method will also be available to malware researcher in possession of the DDP code. Thus, a brute force attack will be always possible.

[7] There is a limit imposed by the key length of the algorithm. If *KS*> the keylength of algorithm A, then the *CF* function will need to perform a digestion in order to produce a key *EK* of that length (i.e. there will be a probability of collisions, as with any one way hashing function).

algorithm (*A*) is yet another way to increase *XBR*. However, according to Schneier, the speed at which each combination can be tried is less important than the number of expected combinations [6]. It should be noted that increasing *XT* might also have an impact (most likely insignificant) in the performance of the DDP.

Anyway, *XT* will depend to a degree on the resources (computer power) available to the malware researcher, so it would make sense for the malware author to use conservative estimates for this value (e.g. consider the use of supercomputers) while calculating *XBR* values.

A few examples[8] should make things clearer. Like before, let us assume that our condition (*CND*) will always be true for simplicity and that the time contributed by functions *CF* and *V* is insignificant. Then let us assume we have 2 DDPs: DDP_1 and DDP_2, which have K_1 and K_2 as their respective data sources, so that:

$K_1 = \{S_1\}, S_1 :=$ A specific time, consisting of hours (0-23), minutes (0-59) and seconds (0-59)
$K_2 = \{S_2, S_3\}, S_2 :=$ An IPv4 IP address (i.e. 4 bytes, each in the range 0-255), $S_3 :=$ a network card MAC address (i.e. 6 bytes, each in the range 0-255).
$XT = 5.88x10^{-7}\,s$

Thus, we would have:

$$XBR_1 = XC * XT = \frac{EKS * XT}{2} = \frac{(24*60*60)*(5.88x10^{-7}\,s)}{2} \approx 0.02s$$

$$XBR_2 = XC * XT = \frac{EKS * XT}{2} = \frac{(256^4)*(256^6)*(5.88x10^{-7}\,s)}{2} \approx 3.55x10^{17}\,s$$

$$= 9.87x10^{13}\,h = 1.12x10^{10} \text{ years}$$

Consequently, we have two very different results. The first DDP (DDP_1) can be brute forced very easily with current computers. However, performing a brute force attack on DDP_2 is well beyond the possibility of most (if not all) malware researchers with current technology and techniques. The *XBR* in the second scenario is longer than the estimated age of the universe: ~10^{10} years [6].

We still have the last option: dictionary attacks. A dictionary attack is one where the obvious (i.e. most expected/probable) combinations for a key are searched for first [6,13]. Given the vast number of data sources that a malware author might choose, malware researchers will need to make use of common sense and context to build dictionaries.

[8] The value for *XT* was obtained by assuming a payload of 10000 bytes an estimated decryption throughput of approximately $1.7x10^{10}$ bytes/second for 3DES (i.e. $10000/1.7x10^{10}=5.88\ x10^{-7}$seconds). The estimate was obtained using the benchmarking options of Openssl and an AMD Athlon 64, 3200+ running on Microsoft Windows XP + SP2. In comparison, the approximate decryption throughputs for Blowfish and AES-128 under the same conditions were: $7.4x10^{10}$ bytes/second and $4.4x10^{10}$ bytes/second respectively. Of course, from the 3 algorithms, 3DES would be an expected choice for malware authors since it is the slowest.

For example, let's take the DDP_2 example reviewed in option 3 (brute force attacks). A malware researcher might ignore some IPV4 addresses and address ranges based on the context of the attack (e.g. broadcast addresses, reserved/private ranges and reserved classes).

But we can go even further if the context allows us. Let us assume that a malware implementing DDP_2 was found in a corporate environment with 5000 machines (most of which had this malware) and that there is strong evidence suggesting that this is a targeted attack (i.e. only one of the machines within this corporation was targeted, but the attacker probably tried to make it look like a general incident to complicate investigations).

It so happens that we only have exactly 5000 possible IP address/MAC pairs in this scenario.

Let *RKS* be the reduced Key Space consisting only of dictionary combinations, where $RKS \leq EKS \leq KS$.

For this particular scenario we have that the estimated *XBR* for DDP_2, with *RKS* =5000:

$$XBR_2 = XC * XT = \frac{RKS * XT}{2} = \frac{(5000) * (5.88x10^{-7} s)}{2} \approx 0.0014s = 1.4ms$$

So, even though a brute force attack might take much longer, with the appropriate knowledge the *RKS* can be significantly smaller than the *EKS*[9]. The obvious solution for malware authors is to avoid using data sources that are not easily related to a context (complicating in this way the creation of dictionaries). Riordan and Schneier analyzed the feasibility of making dictionary attacks impractical by selecting data sources with certain properties and showed that this was possible [1].

Example: An attacker creates a worm that implements a DDP that uses the same data sources as DDP_2. The DDP contains a targeted attack against a specific company (of course, the attacker somehow knows an IP/MAC address combination that works within the targeted company). The worm propagation characteristics (propagation media/ propagation requirements) and the places where it is unleashed give the attacker a very good probability that some computers within the targeted company become infected (before any antimalware company generates a detection signature of course) but in such a way that the target is indistinguishable from other companies (more on this in the next section). By the time the antimalware companies update their signatures and the target company deals with the problem, the damage is done.

If the antimalware companies analyze this particular piece of malware, their analysis might conclude that this thing has N replication characteristics, that due to this, potentially, M computers might have been infected in the world, and that it contains an

[9] Of course, this is not so straight forward in reality. Investigators would need to collect all IP/MAC address pairs to build the dictionary and prepare the dictionary attack using the same code from the DDP. In other situations involving different data sources this might be more difficult and time consuming (e.g. user passwords, number of text documents in the "MyDocuments" folder and email addresses to which the victims send emails regularly). In most cases though, this will still be far more efficient than a brute force attack.

encrypted piece of code that they don't know what it does (because context is unknown, there is not enough information to build a dictionary for an efficient attack on the cryptographic armor).

Even worse, weeks after the incident, the targeted company suffers an attack where some confidential information was stolen; the encrypted payload contained another, completely different, piece of malware that would stay hidden for a few weeks, then it would steal this information from a number of systems and finally destroy itself.

A summary of the suggestions that malware authors might take into account to generate robust DDPs is shown in the next table:

Against:	Robustness requirements:
Cryptanalysis	Use well known, strong, encryption algorithms.
Black Box analysis	Avoid using time dependent data sources for which a reasonable time frame of successful decryption can be obtained. Use at least one time independent data source.
Brute force attacks	Avoid obvious invalid combinations (e.g. use data sources that are independent from each other, don't use a data source that is in the key generation data set for the decryption condition). Use several data sources to increase effective key length (if possible, use data sources that contribute huge numbers of combinations).
Dictionary attacks	Avoid context dependent data sources.

4. PRACTICAL USES OF DDPS FOR MALWARE AUTHORS

So far, it has been demonstrated that it is possible to implement a DDP in malware and that some implementations are impossible to deal with traditional analysis techniques, including reverse engineering [3]. We have also shown in the previous section the requirements for a robust DDP.

However, there is still the question of how practical can a robust DDP be for an attacker. In other words: Are robust DDPs really usable by malicious individuals or groups?

The answer in short is yes. There are various scenarios where the use of DDPs is practical. Furthermore, there are a few scenarios where their use is nearly ideal for criminals. So, let us take a look at these scenarios.

Targeted attacks

The more (independent) data sources[10] an attacker uses within a DDP, the more restricted the decryption conditions become. But for attackers with specific targets this is not an issue. On the other hand, using robust DDPs for targeted attacks allow attackers to complicate any investigation significantly.

For example, in a crime involving malware using DDPs, even if investigators find the malware in the machine and suspect it to be involved in the crime, unless they are able to decrypt the payload, there is no evidence that it was involved in the incident (let alone getting any useful information on how or by whom the attack was prepared and executed). Several cases of convicted malware authors and new regulations give these people more incentives to be more careful. In fact, this is probably one of the reasons why there is an increase in targeted attacks (more benefits with potentially less risks). It is a known fact that targeted attacks for committing frauds are on the rise, and DDPs can be (or are already) a valuable tool for these criminals. The AWPG has interesting statistics on the use of some malware used for targeted attacks [14].

Targeted attacks with unknown targets

As long as the attackers are able to disseminate their DDP enabled malware at several places and choose their data sources carefully, the identification of the target, for anyone other than the target itself or someone close to it, might be impossible in practice.

An example at the end of the previous section showed how DDPs could be used to confuse a malware researcher (DDP_2). In that example, unless investigators are able to relate the previous worm infection to the incident being investigated (or at least be suspicious), there is no way of generating a useful dictionary attack.

Only the victims or investigators working in the victim's case might be the only people in a position to actually identify a useful context from which a proper dictionary attack can be conducted. Everyone else (including antimalware companies) won't be able to identify a proper context by themselves.

If only the target (or someone close) might be able to identify a proper context to produce a useful dictionary, for a dictionary attack, the possibility of recovering the payload from a robust DDP via a dictionary attack might not be within the reach of external experts (i.e. cryptanalysts, independent malware researchers and antimalware companies). Only if the target (victim) is able to identify this situation and if it is then willing to communicate this to the right people could this result in a successful dictionary attack. In a worst case scenario, the best that any entity without knowledge of the specific context can hope for is a brute force attack.

It is no secret that many companies prefer to hide the fact that they have been attacked (were it through malware or by other means of intrusion). Some are now required by law to disclose some information regarding such incidents in some countries. But take a look at this hypothetical scenario:

[10] Note that not too many independent data sources are needed to create robust DDPs. E.g. the last example in section 3 only required 2 of them.

A company suffers an incident and during an internal investigation detects a malware with a robust DDP. Without the resources (tools/people/knowledge) they can't analyze it properly. Would they disclose this code to external experts to find out if it is related to the incident, even if there is no proof of it (and hence probably no obligation to do it)? What if the analysis reveals a different, much worse incident? What if the DDP was found within a "legal" piece of software for which they have a user licence but, as most commercial licences specify, they don't have the right to reverse engineer it (let alone asking someone else to do it)?

More general attacks and the key complexity problem

Previously we noted a potential problem with DDPs: in its simplest form, they are useful for specific, targeted, attacks but not so useful for more general attacks. Choosing data sources so that the attack is feasible against a wider range of victims could contravene our previously defined requirements for a robust DDP, making black box analyses or brute force attacks feasible.

There are at least 2 possibilities that allow malware authors to increase the effective key space while also allowing some flexibility to widen the attack[11]. These are: key combinations and controlled data sources.

Riordan and Schneier included thresholding in their analysis of further constructions [1]. A thresholding scheme would allow an attacker to specify *K* data sources so that only *N* out of *K* would be required to decrypt the payload. This would work, but the effective key space, *EKS*, would then depend on the weakest combination of any possible *N* data sources. Since a malware researcher will always have access to the data source set *K* (as previously shown), she/he would obviously choose the combination of data sources that provides the least number of key combinations.

For example, if the data sources in *K* are the current system time (in minutes), the IP address, the MAC address, and the day of the month in the system date, with a threshold scheme using 2 out of any of the 4 specified sources, a malware researcher can execute a brute force attack easily by choosing the system time and the day of the month.

Still, an attacker can go beyond the basic thresholding concept and specify exactly which *N* out of the *K* data sources have to be used by trying all possible combinations in the decryption routine. The idea is that, with the exception of small increase in the time it takes to attempt each decryption, the DDP will still work as expected, whereas a malware researcher would need to brute force all combinations[12].

An example should help clarify this. To be consistent with our last example in the previous section:

[11] Even with this flexibility, it should be noted that, by the very nature of DDPs, a completely general attack is impossible. There will exist always a certain context that will restrict the potential number of targets.

[12] As long as the malware author complies with the requirements of a robust DDP, there should be no easy way to identify which *N* out of the *K* data sources are being combined to produce the correct key.

Let $K = \{S_1, S_2, S_3\} : S_1 :=$ A specific time, consisting of hours (0-23), minutes (0-59) and seconds (0-59), $S_2 :=$ An IPv4 IP address (i.e. 4 bytes, each in the range 0-255), $S_3 :=$ a network card MAC address (i.e. 6 bytes, each in the range 0-255).
Let $XT = 5.88x10^{-7}\, s$.
Let $N = \{S_3\}$.

For the decryption routine, the malware would need to try as many decryptions as there are possible combinations of sets with 1, 2 and 3 elements *S* so that $S \subseteq K$.

Let *CB(K)* be the number of combinations of elements $S \subseteq K$, of all possible sets with 1 to *F* elements, where *F* is the cardinality of set *K* (|*K*|)[13]:

$$CB(K) = \sum_{j=1}^{F} \frac{F!}{j!(F-j)!}$$

In this case we have then $CB(K) = \frac{3!}{1!(3-1!)} + \frac{3!}{2!(3-2)!} + \frac{3!}{3!(3-3)!} = 3+3+1 = 7$

So, the decryption routine within the DDP would need to attempt 7 decryptions:
$V(D_A(EP,(CF(\{S_1\}))))$
$V(D_A(EP,(CF(\{S_2\}))))$
$V(D_A(EP,(CF(\{S_3\}))))$
$V(D_A(EP,(CF(\{S_1,S_2\}))))$
$V(D_A(EP,(CF(\{S_1,S_3\}))))$
$V(D_A(EP,(CF(\{S_2,S_3\}))))$
$V(D_A(EP,(CF(\{S_1,S_2,S_3\}))))$

Then, in this example, each decryption attempt by the DDP will take $CB(K) * XT = 7 * 5.88x10^{-7}\, s = 4.116x10^{-6}\, s$. Not a big deal. The decryption time increases linearly by a factor of *CB*(*K*).

However, things for a malware researcher get considerably more complex. Let us calculate the *XBR* for each of the 7 combinations that she/he needs to try, in order to recover the key with a brute force attack:

$$XBR_{S_1} = XC_{S_1} * XT = \frac{(24*60*60)*(5.88x10^{-7}\, s)}{2} \approx 0.02s$$

[13] This formula is derived from the well known formula, $\frac{n!}{r!(n-r)!}$, that is used to obtain the combinations of *r* objects from a group of *n* objects. Some clear and simple explanations on the uses of permutations and combinations in cryptography are provided in [5].

$$XBR_{S_2} = XC_{S_2} * XT = \frac{(256^4) * (5.88x10^{-7} s)}{2} \approx 1262.72s$$

$$XBR_{S_3} = XC_{S_3} * XT = \frac{(256^6) * (5.88x10^{-7} s)}{2} \approx 82.75x10^6 s$$

$$XBR_{S_1.S_2} = XC_{S_1.S_2} * XT = \frac{(24 * 60 * 60) * (256^4) * (5.88x10^{-7} s)}{2} \approx 109.09x10^6 s$$

$$XBR_{S_1.S_3} = XC_{S_1.S_3} * XT = \frac{(24 * 60 * 60) * (256^6) * (5.88x10^{-7} s)}{2} \approx 7.14x10^{12} s$$

$$XBR_{S_2.S_3} = XC_{S_2.S_3} * XT = \frac{(256^4) * (256^6) * (5.88x10^{-7} s)}{2} \approx 3.55x10^{17} s$$

$$XBR_{S_1,S_2.S_3} = XC_{S_1,S_2.S_3} * XT = \frac{(24 * 60 * 60) * (256^4) * (256^6) * (5.88x10^{-7} s)}{2} \approx 3.07x10^{22} s$$

The DDP can generate the correct decryption key using only data source S_3, ($XBR_{S_3} \approx 82.75x10^6 s \approx 2.6$ *years*), but the malware researcher, in a worst case scenario, won't know this. From her/his point of view:

$$XBR = XBR_{S_1} + XBR_{S_2} + XBR_{S_3} + XBR_{S_1,S_2} + XBR_{S_1,S_3} + XBR_{S_2.S_3} XBR_{S_1,S_2.S_3}$$

This results in a clearly unfeasible brute force attack by current standards. All that a malware researcher could probably do is try the combinations with a manageable *XBR* (i.e. the first 2). But look at this scenario: a DDP using 20 independent data sources in set *K*, each of which has an estimated *XBR* of 6 months on average. Where would the malware researcher start? If she/he tries all 20 data sources it would take, on average, around 10 years to recover the payload.

Data sources controlled by the attacker constitute a second possibility for the attacker. If the attacker controls a specific data source (or at least can predict its behaviour reliably enough), she/he could then use it as part of the set *K*. There are a few advantages:

- Some flexibility to choose/change targets after the malware has been distributed.
- Many controlled data sources could provide, on their own, a huge number of combinations without restricting the attacker to a specific target (e.g. a data source consisting of the first 128 bytes of payload from an ICMP packet, with a DDP that sniffs network traffic).

There is also a disadvantage:

- If the malware is discovered, and since malware authors will always be able to identify the data sources within the *K* set, a scheme to monitor this data source could be set up and it might be able to trace the attacker when a successful decryption is performed.

Decoys and the justified purpose problem

Keys are not the only place where malware authors can play with combinations. Nothing prevents malware authors from including several supposedly encrypted payloads; the

DDP would then try each key with each payload. While this could be considered as another method of increasing brute force attack time, it has other advantages that deserve a separate discussion.

We already know that malware researchers will always have access to most DDP elements and that they should always be able to identify the presence of a DDP. However, is this enough to classify a DDP as malicious?

Riordan and Schneier mentioned one legitimate purpose of programs using environmental keys (mobile agents); they also mentioned the possibility of nesting [1]. There is another legitimate application of DDPs: anti piracy schemes.

Let us imagine that a company creates a game and encrypts every level within this game. The game uses some game environment variables as data sources for decrypting each next level, so that it has some certainty that the level was completed correctly. Each level has (different) routines that verify that the game copy installed is legitimate (e.g. connect to a server of the game developer and check that the registration code is valid).

The problem for pirates is this: they can reverse engineer the first level and crack the protection in there, but, if a robust DDP is used to protect each new level, they will need to play the whole game and get to the last level to be able to sell pirated copies of a complete game. The security assumption by the vendor is this: if the game is hard enough, by the time that pirates are able to crack all levels there will be too many more people that finished legitimate games as well (i.e., they had to buy legitimate copies to play because pirates couldn't offer a complete copy), and old games are not so attractive. Also, if pirates risk selling incomplete games, people will eventually realize this, and for once they might believe the developers when they say that quality of pirated software is not the same. If you doubt how far people can go to protect their copyrighted material [15] has a detailed account of the 2005 Sony CD copy protection scandal.

So, with a few legitimate applications, how can a malware researcher claim, beyond reasonable doubt, that anything with a DDP is malicious?

This is important. Antimalware companies were able to identify several polymorphic engines in the past and were still capable of unmistakably determine that the protected code was malware because they could analyze what the payload was doing. But this is not the case with robust DDPs. Not within a reasonable time at least.

This is how a malware author could install a trojan (or any other type of malware) in a victim's company and still (probably) avoid jail: first, a legitimate application is created that contains a robust DDP with, say, 5 different encrypted payloads.

The attacker then tricks its victims into installing this "useful" piece of software (by selling it to them, giving it for free or by any other means). For an example of how easy employees can be tricked into running software refer to [16].

2 of the payloads contain useful data (one is the malware, the other is a copy protection scheme similar to the one described above which is consistent with the license of use included with the program). The other 3 payloads contain only random data (decoys).

The malware is eventually decrypted and installed into some machines where the program was installed. An incident occurs and investigators are called into action. The investigators locate the malware and suspect of the program's DDP, but can't prove that the malware was actually installed by the "useful" program. Luckily enough, they reside in a country where law enforcement can demand that decryption keys are handed over during a crime investigation (for a discussion of such requirements by law enforcement in England, refer to [17]).

Letting aside the issues with "probable cause", let us assume that investigators are able to get law enforcement involved which in turn provides them with a warrant for any encryption/decryption keys for any encrypted parts within the "useful" program. In that case, the attacker simply gives away the key for the copy protection scheme (and then probably sues the victim for defamation and other damages generated by the disclosure of the proprietary copyright protection scheme).

How can the victim, the investigators or the law enforcement probe that the malware still lies within the other 4, supposedly decoy, payloads beyond reasonable doubt when a brute force attack is unfeasible?

Malware unknown updates and updates from unknown sources

Last, but not least, let us analyze a problem that has been present with some worms and viruses.

Several variants of a number of worms and viruses included within their code locations (IP addresses or server names) from where they could download updates, in order to be ahead of the antimalware companies. So far many antimalware companies have been able to thwart further updates by analyzing quickly each piece of malware and identifying these locations to shut them on time.

With DDPs malware authors have two choices: encrypt predefined updates that will trigger a few hours or days after the infection, or encrypt the update routine along with the locations containing the updates (which would give attackers even more flexibility).

The data sources selected in this case would need to be such that they are flexible enough to successfully update an important number of previously infected machines and yet still be able to prevent antimalware companies from accessing the (encrypted) updates, or from determining the location for the next update, on time.

The keys don't need to be too complex. By the time that enough machines have been updated the secret won't be important anymore. All that malware authors need are a few hours or days of advantage, and with all previously mentioned techniques it is left to the readers to try some numbers and examples to satisfy themselves that this is in fact possible.

5. ANALYZING MALWARE THAT USES A DDP

Although DDPs, when used for malicious purposes, can pose a significant challenge for malware researchers and computer forensic investigators, that doesn't mean that all cases will be equally difficult.

As was previously discussed, malware authors need to consider several requirements in their implementations, in order to produce robust DDPs.

Investigators and researchers should note that there is always the possibility of an efficient dictionary attack, as long as the correct context can be identified. Setting up appropriate communication channels with potential victims of such attacks would be valuable.

The increase in targeted attacks also means that tools whose design relies on a blacklist approach (i.e. most antimalware tools currently available) won't be useful to prevent or even identify such attacks, especially if they include robust DDPs. But these shouldn't be surprising news, since it was proved a long time ago by Cohen that only controls capable of limiting sharing, transitivity and programming could possibly stop computer viruses and malware effectively [18]. White list based security controls[14] are the closest thing we have to deal with malware implementing DDPs right now.

This trend suggests that people investigating malware with robust DDPs should possess at least some skills and knowledge on cryptography, cryptanalysis and reverse engineering.

To facilitate the analysis of such malware the following suggestions might be helpful:

- To identify the existence of a DDP within a program, locate a section of code that would transfer control to a part of the program that is currently not executable and looks random[15].
- After a DDP has been located, concentrate your efforts on identifying the unencrypted elements: the data sources (*K*) and any decryption condition functions (*CND*) with their associated data sources (S_S).
- Calculate the effective key space (*EKS*) and determine if a black box analysis is possible.
- If a brute force attack or a dictionary attack is required, identify also the decryption function and algorithm (D_A), the construction function (*CF*), the Verification function (*V*), the verification code (*VC*).
- Try to identify the possible context to produce a reduced key space (*RKS*). Try to do some educated guesses if necessary and perform a dictionary attack if the size of *RKS* still makes it feasible. Always use the same D_A, *CF*, *V* and *VC* included in the DDP (you never know when a malware author will test a variation of what looks like a well know cryptographic algorithm).
- If dictionary attacks seem unfeasible, and if the size of *EKS* is still reasonable, attempt a brute force attack.
- As a last resort, if everything else fails or seems unfeasible: think!
    - Look at any oddities in the implementation of the DDP
    - Communicate with peers
    - Review the latest advances in cryptanalysis and DDP analysis

[14] White lists based security controls by default block everything that has not been explicitly authorized at some level (e.g. firewalls at network level, security shells at O.S. level). In contrast, black list based security controls block preciously known unauthorized actions and behaviour. Hence, they depend on updates and timely analysis of new attacks to be effective.

[15] Good encryption algorithms should produce encrypted outputs that are indistinguishable from random sequences [6].

- Try to do things that might induce the malware author to reveal himself
- Use alternate information streams (search the Internet, maybe you find someone in an IRC channel discussing the same DDP you are facing).

Never loose hope, after all, we still need to see if someone ever calls malware analysts to take a look at malware implementing DDPs. It is yet to be seen how many criminals will employ these techniques. Cryptography has been available for protection purposes for a number of years now, nevertheless, many still don't use it, and there is no guarantee that criminals will be more careful take advantage of it in big numbers.

6. CONCLUSIONS

Dynamic Decryption Procedures in malware, which use the concept of Environmental Key Generation as its foundation, are relatively easy to implement and have been known to researchers for a number of years. Therefore, it shouldn't be long before we see malware using similar schemes in the wild (if it has not happened already). Also, there are several scenarios where their use by criminals within malware is not only practical, but also gives them important advantages.

Malware researchers won't be the only people interested in the development DDPs. Forensic computing investigators will also be interested in the use of these routines within malware since, as was mentioned before, they are particularly useful to commit computer related crimes while hiding traces (e.g. when used within trojan horses in targeted attacks). Developers researching legitimate uses of DDPs will also be paying attention to related techniques.

With current trends showing an increasing number of targeted attacks using malware, it might be necessary to include cryptographic training as part of a malware researcher's formation, and to develop specific cryptanalytic techniques for encrypted malware. It should not be assumed anymore that complete analyses of malware will always be completed in a reasonable time.

Reverse engineering will still remain as one of the main techniques to analyze malware. However, reverse engineering alone won't be enough to successfully analyze malware implementing DDPs. It is possible that many targeted attacks could only be detected or analyzed on site. Also, due to the limits of current antimalware technology, antimalware companies might be motivated into extending their services for on-site analysis of targeted/custom malware (some of which might potentially have DDPs). At the very least, the way that companies developing antimalware products work will need to change drastically in the future; pretending that the problem does not exist or that DDPs are not practical will just make things worse and doesn't help in any way.

The use of white list based security controls might not only be more convenient to prevent these threats, they are probably the only solution right now that could be effective if implemented and configured correctly. From a preventive point of view, It doesn't matter if a program with a DDP can't be identified/detected as long as any malicious payload is prevented from doing its job; some white list based security controls (such as security shells) attempt to tackle this problem.

7. REFERENCES

[1] J. Riordan and B. Schneier, *Environmental Key Generation towards Clueless Agents*, Mobile Agents and Security, G. Vigna, ed., Springer-Verlag, 1998, pp. 15-24.

[2] A. Young, and M. Yung, *Malicious Cryptography: Exposing Cryptovirology*, Wiley Publishing, New Jersey, 2004.

[3] E. Filiol, *Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the BRADLEY virus*, Proceedings of the 14th EICAR Conference, 2005.

[4] A. Hirt and J. Aycock. *Anonymous and Malicious*. Conference proceedings of Virus Bulletin, October 12-14, 2005, pp. 2-8.

[5] R. E. Lewand, *Cryptological Mathematic*s. 2nd Edition. The Mathematical Association of America, Washington, 2000.

[6] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*, Wiley, New York, 1996.

[7] Skape, *Understanding Windows Shellcode*, 2003, http://www.nologin.org/Downloads/Papers/win32-shellcode.pdf.

[8] F. Raynal, *Malicious Cryptography*, Part Two, SecurityFocus, 16 May 2006, http://www.securityfocus.com/infocus/1866.

[9] R. Kuster, *Three Ways to Inject your Code into another Process*, Section III, 21 August 2003, http://www.codeproject.com/threads/winspy.asp#section_3.

[10] P. Ferrie and P. Ször, *Zmist Opportunities*, Virus Bulletin, March 2001

[11] P.Ször and P. Ferrie, *Hunting for Metamorphic*, Symantec, 2003, http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf.

[12] S. J. Shepherd, *Cryptography: Diffusing the Confusion*, Research Studies Press, Baldock, 2001.

[13] A. J. Menezes et. al., *Handbook of Applied Cryptography*. CRC Press, Boca Raton, 1997.

[14] APWG, *Phishing Activity Trends Report*. April 2006. Anti-Phishing Working Group, http://www.antiphishing.org/reports/apwg_report_apr_06.pdf.

[15] Wikipedia, *2005 Sony CD copy protection scandal*, http://en.wikipedia.org/wiki/2005_Sony_CD_copy_protection_controversy.

[16] W. Sturgeon, *Proof: Employees don't care about security*, silicon.com, 16 February 2006, http://software.silicon.com/security/0,39024655,39156503,00.htm.

[17] T. Espiner, *British legislation to enforce encryption key disclosure*, CNet News, 18 May 2006, http://news.com.com/2100-7348_3-6073654.html

[18] F. Cohen, *A Short Course on Computer Viruses*, 2nd Ed., John Wiley & Sons, New York, 1994.

APPENDIX A – PROOF OF CONCEPT CODE

The following code illustrates an implementation of a DDP. The effective keyspace in this particular example is not very long; the only sources of information are certain fields of the date/time registered by the machine (hour, month, year).

The encrypted payload consists of a shellcode that will execute a continuous ping on the local host (IP address 127.0.0.1) once it has been successfully decrypted. This particular example uses blowfish as the cryptographic algorithm.

The example is specific to Windows and requires the use of Openssl crypto libraries and windows libraries (i.e. linking with libcrypto and libkernel32 libraries) to compile. The main loop in the routine is the do…while section; it will awake every 10 seconds and try to decrypt and execute the shellcode which will only happen if the year is 2005, the month is December and the hour is 21 (i.e. 9pm, with any value for minutes and seconds).

The key is not included within the program body (it is derived from the predefined date/time information, which is then XORed with the values 0x0F, 0x20, 0x6A, 0x04, 0x10, 0x0D, 0x6E and 0x08 to add some confusion).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <openssl/blowfish.h> /*Openssl Blowfish encryption functions*/
#include <openssl/md5.h>      /*Openssl MD5 hashing functions*/
#include <windows.h>           /*Required for use with Sleep()*/

#define SCODE_SIZE 172+8
void (* shell)();
/* will use gmtime to derive the key
struct tm {
    int tm_sec;         seconds after the minute (from 0)
    int tm_min;         minutes after the hour (from 0)
    int tm_hour;        hour of the day (from 0)
    int tm_mday;        day of the month (from 1)
    int tm_mon;         month of the year (from 0)
    int tm_year;        years since 1900 (from 0)
    int tm_wday;        days since Sunday (from 0)
    int tm_yday;        day of the year (from 0)
    int tm_isdst;       Daylight Saving Time flag
    }; */

int main(int argc, char *argv[])
{                                       /* key, IV must be 8 byte blocks */
   time_t rawtime;
   struct tm * gmt;
   int cont;
   unsigned char decryptedPayload[SCODE_SIZE+10];
   unsigned char keyTime[8];
      /*We will only use 4 bytes: 1 byte frome the hour, 1 from the month and 2
        from the year, and then duplicate to get 8 bytes for XORing to get the key:
        DD MM Y1 Y2 DD MM Y1 Y2*/
   unsigned char key[8];       /*Blowfish Derived key from keyTime XOR keyXOR*/

   unsigned char ivec[8]={0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0};  /*Initial vector*/
   unsigned char decryptedMD5Hash [16];
   unsigned char scodeMD5Hash[]={0x81,0xED,0xD3,0x62,0xC6,0xE3,0x89,0x4F,0xD5,0xA5,
```

```
                              0x6C,0x95,0xCE,0x55,0x25,0xE0};
unsigned char keyXOR[8]={0x0F,0x20,0x6A,0x04,0x10,0x0D,0x6E,0x08};

BF_KEY sched;

unsigned char encryptedPayload[SCODE_SIZE+10] = {
0xD7,0x5A,0x9A,0x75,0x12,0x8C,0x44,0xC0,0xC3,0xEB,0xBC,0x32,0x86,0xD8,0xB2,0xD4,
0xAF,0xB6,0x36,0xB4,0x24,0x4D,0x7F,0x2F,0x6D,0x70,0x32,0xA7,0xDD,0xCC,0xAB,0x0C,
0x5F,0x9E,0x79,0x26,0x70,0xA5,0x6F,0xAC,0xB4,0xB6,0x71,0x13,0xBE,0xDB,0xDA,0x0F,
0xAD,0x1C,0x10,0x8D,0x31,0x18,0x34,0x5E,0xEA,0x74,0xC7,0xD2,0x55,0x86,0xB8,0xFA,
0x40,0x47,0x0E,0x78,0xA5,0xAB,0x45,0x8E,0x08,0x0C,0xDA,0x68,0xD6,0x42,0x54,0x99,
0xE6,0x54,0xA1,0xFC,0x48,0xAB,0xBA,0x9E,0x62,0x6D,0x52,0x3C,0x49,0xCA,0x2A,0xAB,
0x63,0x0B,0xCD,0x1A,0x3C,0xF7,0x15,0x2E,0xB1,0x4D,0x15,0x11,0xEA,0x78,0x27,0x3B,
0x33,0x81,0xD3,0x9D,0x8D,0x9B,0xE7,0xBB,0x0C,0xC5,0x97,0x8C,0x8E,0x38,0x49,0xE7,
0xFD,0xAB,0x13,0x28,0x9F,0x45,0xAC,0x1C,0xE0,0x62,0xC3,0x82,0x47,0xB8,0x4A,0xA3,
0xB0,0x14,0x2C,0xFA,0xC4,0xE1,0x51,0x6A,0x15,0x77,0xDF,0xA0,0x3F,0x24,0x98,0x36,
0x7B,0x2D,0xC2,0x22,0x82,0x76,0x9A,0xBC,0x81,0xBF,0x09,0x01,0x9C,0xBE,0xB2,0x54,
0xEB,0xF2,0xFC,0x7D,0xD1,0x77,0x44,0xDB,0x00,0x00,0x00,0x00,0x00,0x00};

memset(decryptedPayload,'\0',SCODE_SIZE+10);

do{
   Sleep(10000);  /*This avoids overusing the CPU*/
   time(&rawtime);
   gmt=gmtime(&rawtime);
   keyTime[0]=((unsigned char) gmt->tm_hour);
   keyTime[1]=((unsigned char) gmt->tm_mon);
   keyTime[2]=((unsigned char) (gmt->tm_year));
   keyTime[3]=((unsigned char) ((gmt->tm_year)>>8));
   keyTime[4]=((unsigned char) gmt->tm_hour);
   keyTime[5]=((unsigned char) gmt->tm_mon);
   keyTime[6]=((unsigned char) (gmt->tm_year));
   keyTime[7]=((unsigned char) ((gmt->tm_year)>>8));

   key[0]=keyTime[0]^keyXOR[0];
   key[1]=keyTime[1]^keyXOR[1];
   key[2]=keyTime[2]^keyXOR[2];
   key[3]=keyTime[3]^keyXOR[3];
   key[4]=keyTime[4]^keyXOR[4];
   key[5]=keyTime[5]^keyXOR[5];
   key[6]=keyTime[6]^keyXOR[6];
   key[7]=keyTime[7]^keyXOR[7];

         /*Set Blowfish Key*/
   BF_set_key(&sched,8,key);

   BF_cbc_encrypt(encryptedPayload, decryptedPayload,    /*Decrypt payload*/
        SCODE_SIZE,&sched,ivec,BF_DECRYPT);
   memset(decryptedMD5Hash,'\0',16);
   MD5(decryptedPayload+8, SCODE_SIZE-8,decryptedMD5Hash);
} while (((unsigned long long) *decryptedMD5Hash!=
  (unsigned long long) *scodeMD5Hash)||
  ((unsigned long long) *(decryptedMD5Hash+8)!=
  (unsigned long long) *(scodeMD5Hash+8)));

printf("CORRECT KEY =%.2X%.2X%.2X%.2X%.2X%.2X%.2X%.2X\n",
      (unsigned char) key[0],(unsigned char) key[1],
      (unsigned char) key[2],(unsigned char) key[3],
      (unsigned char) key[4],(unsigned char) key[5],
      (unsigned char) key[6],(unsigned char) key[7]);

printf("Encrypted SCODE is (salted): \n");
for (cont=0;cont<SCODE_SIZE+10;cont++){
   printf("%.2X ",(unsigned char)encryptedPayload[cont]);
   if (((cont+1)%16)==0){
      printf("\n");}
}

printf("\nDecrypted SCODE (without salt) is: \n");
for (cont=8;cont<SCODE_SIZE+10;cont++){
   printf("%.2X ",(unsigned char)decryptedPayload[cont]);
```

```
      if (((cont+1)%16)==0){
         printf("\n");}
   }

   shell= (void *) (decryptedPayload+8);  /*skip salt and jump to decrypted code*/
   shell();
   return 0;
}
```