



Office de la Propriété  
Intellectuelle  
du Canada

Un organisme  
d'Industrie Canada

Canadian  
Intellectual Property  
Office

An agency of  
Industry Canada

CA 2372034 A1 2003/08/14

(21) **2 372 034**

(12) **DEMANDE DE BREVET CANADIEN  
CANADIAN PATENT APPLICATION**

(13) **A1**

(22) Date de dépôt/Filing Date: 2002/02/14

(41) Mise à la disp. pub./Open to Public Insp.: 2003/08/14

(51) Cl.Int.<sup>7</sup>/Int.Cl.<sup>7</sup> G06F 12/14

(71) Demandeur/Applicant:  
CLOAKWARE CORPORATION, CA

(72) Inventeurs/Inventors:  
CHOW, STANLEY, CA;  
JOHNSON, HAROLD, CA;  
MAIN, ALEC, CA

(74) Agent: GOWLING LAFLEUR HENDERSON LLP

(54) Titre : SYSTEME DE PROTECTION PERMETTANT DE CONTRECARRER LES ATTAQUES PAR SURCHARGE  
DES MEMOIRES TAMPONS ET PAR CODES ETRANGERS AU MOYEN DU CODAGE

(54) Title: FOILING BUFFER-OVERFLOW AND ALIEN-CODE ATTACKS BY ENCODING



# FOILING BUFFER-OVERFLOW AND ALIEN-CODE ATTACKS BY ENCODING

STANLEY CHOW, HAROLD JOHNSON, AND ALEC MAIN

**ABSTRACT.** Despite the deployment of various kinds of defenses, worm programs, viruses, and other kinds of remote attacks over the Internet continue to cause damage, disrupt traffic, and otherwise hobble computer- and communications-based productivity.

This invention deploys a new kind of diversity-based defense, called the *Babel defense*, against one of the most problematic kinds of attack: *buffer overflow* attacks. After considering various alternatives, including other kinds of diversity-based defenses, and why they are relatively ineffective, we explain how the new kind of defense shuts down buffer overflow attacks effectively, and also provides strong protection against hostile executable code from external sources.

The new defense is not dependent on historical information, and therefore is highly effective even against brand-new attacks. It even provides an effective defense against *insider*-initiated attacks.

In its broad outline, this is a straightforward extension of the authors' co-pending patent application entitled **Computer System Protection by Communication Diversity**[6]. However, its actual deployment is rendered complex by certain intricacies of real software on real systems. This disclosure provides methods for dealing with such complications.

## CONTENTS

|                                                                     |    |
|---------------------------------------------------------------------|----|
| 1. Context of, and Need For, the Invention                          | 2  |
| 1.1. Buffer Overflows: The Highest Risk Attacks                     | 2  |
| 1.2. Previous Buffer-Overflow Prevention Schemes                    | 2  |
| 1.3. Diversity-Based Defenses                                       | 3  |
| 2. Overview of The Babel Defense                                    | 5  |
| 3. The Method: Foiling Buffer-Overflow and Alien-Executable Attacks | 5  |
| 3.1. The Key Observation                                            | 5  |
| 3.2. Making Buffer Overflows Meaningless                            | 6  |
| 3.2.1. Partitioning Code and Data Memory                            | 6  |
| 3.2.2. What is a Buffer?                                            | 7  |
| 3.2.3. What to Encode and How to Encode It                          | 8  |
| 3.2.4. Why We Encode: Effect on Buffer-Overflow Attacks             | 11 |
| 3.2.5. Can the Encoding Defense be Penetrated?                      | 13 |
| 3.3. Other Aspects of Protection                                    | 14 |
| 3.3.1. Handling Application Boundaries                              | 15 |
| 3.3.2. Identifying Codings at Run-Time                              | 17 |
| 3.3.3. Loaders and Interpreters: Handling Alien Executable Code     | 21 |
| 4. CLAIMS                                                           | 24 |
| References                                                          | 27 |

## 1. CONTEXT OF, AND NEED FOR, THE INVENTION

'Remote-control' attacks performed over the Internet, either by viruses, worm programs, and other 'canned' *exploits*<sup>1</sup> of system flaws[20], or the use of such flaws in a remote attack by a *cracker*<sup>2</sup>, have become very serious problems for computer systems connected to the Internet. There are many kinds of defense, ranging from identifying *malware*<sup>3</sup> or other remote attacks and preventing their completion[23], to profiling expected behavior, and then (once unexpected behavior is detected) limiting the damage by shutting down part or all of the system[23], to 'sand-box' approaches in which imported software runs in a limited sub-environment[15], to rule-based approaches based on military security systems[21].

The fact that such remote attacks continue to cause great damage, despite the use of firewalls, virus-detection software[18], and all of these other defensive measures, clearly demonstrates that *a stronger form of defense is needed*.

**1.1. Buffer Overflows: The Highest Risk Attacks.** A very high proportion of the high-risk vulnerabilities reported concern *buffer overflows*. Buffer overflow vulnerabilities result from failure to limit input lengths, so that input can overwrite memory used for other purposes. For example, the ICAT ranking of top ten vulnerabilities, as of January 18, 2002, includes three buffer-overflow problems, of which two concern Microsoft's Internet Information Services (IIS<sup>TM</sup>) used for web-hosting, and one concerns the the mapper from Simple Network Management Protocol (SNMP) to Distributed Management Interface (DMI) mapper for Solaris<sup>TM</sup>, widely used for servers[14]. Plainly, these vulnerabilities concern areas where substantial volumes of commercial Internet traffic could be compromised.

In the open-source world, of the top ten security alerts listed at the Debian<sup>4</sup> home page[12], as of January 18, 2002, three also concerned buffer overflow vulnerabilities!

Not only are buffer overflows a wide-spread and highly persistent problem, they also represent extremely serious risk, because an attack can use such a vulnerability to run arbitrary code with the privileges of the attacked application. Since many applications run with very high levels of privilege (using such facilities as the *setuid* or *setgid* directory flags of Unix<sup>TM</sup> or GNU/Linux), an exploit's code can often do practically arbitrary damage, and can obtain any information available on the attacked system.

**1.2. Previous Buffer-Overflow Prevention Schemes.** The kinds of defenses which have been deployed previously against buffer overflow attacks are conditioned by the sources of such vulnerabilities, which are:

1. System data-input facilities which do not specify a limit on length.
2. Systems implementation languages such as C and C++, which do not mandate array bounds checking (nor require that array bounds be declared), or which

<sup>1</sup>An *exploit* is a form of attack, often embodied in a software entity such as a virus or worm program, which makes use of a system vulnerability in order to perform some action not intended by the system's designers.

<sup>2</sup>A *cracker* is a system-savvy person who employs the knowledge in hostile ways to the detriment of the system's intended use.

<sup>3</sup>*Malware* is a general term referring to *any* kind of software entity — directly executable, executable by an interpreter, or non-executable — whose purpose is to harm or to obtain unauthorized access to a computer system, typically with no human intervention in its operation. It includes, but is certainly not limited to, computer viruses and worm programs.

<sup>4</sup>A project providing an entirely open-source GNU/Linux distribution.

use character strings or other data terminated by a sentinel value (e.g., for C or C++, the *null* character), rather than by a separate count.

3. Absence of hardware protection against overwriting of code, or against execution of data (as in Intel's IA-32, where the memory-protection hardware does not distinguish readable data from executable code); or failure of a system or application to employ such hardware protections when available.

There are compilers for C and C++ which enforce array-bounds checks[2]. Such checks eliminate many of the array vulnerabilities, but not all: C and C++ do not require arrays to have declared bounds, and idiomatic C or C++ code often uses a varying pointer rather than an array and an index. The fully dynamic approach, as in [2], provides almost complete coverage if the library has been completely converted to this form (which, depending on the library, might be a large project in itself), but even then, there are idiomatic C usages (such as zero-length, unsized, or unit-length arrays at the ends of allocated structures to allow for flexible arrays) which are hard to cover fully in this way. (Moreover, the performance penalty for the use of full bounds checking, as in [2], is sufficient to forbid general deployment, except as a debugging technique — between 75% and 200% overhead in code size and execution time.) Although C++ permits the definition of arrays with checked bounds, the ISO-standard C++ library contains a `vector<>` template for which the default indexing (`operator[]`) implementation does *not* check the bounds! (Whatever the merits or otherwise of this design decision, unless programmers use the ISO standard library with full cognizance of this hazard, it will virtually guarantee an ongoing supply of buffer-overflow flaws to exploit.)

Other kinds of protections include the use of *canaries*<sup>5</sup>: special data values placed at strategic points in memory and checked for integrity. The difficulty is that it cannot be guaranteed that any use of canaries light enough to permit good performance will also capture attacks against which they have not been tested. That is, canaries have a *coverage* problem: only parts of the data are protected. Canaries, by their very nature, provide 'spot' checks.

The bottom line is that dynamic checking in compilers and memory management facilities can only provide partial coverage, or nearly full coverage with such high performance penalties that the protection is relegated to the status of a debugging tool. Moreover, history plainly teaches that, when such defenses are deployed, their vulnerabilities (any points which they do *not* cover) will become known all too quickly.

**1.3. Diversity-Based Defenses.** Cohen[7] and Forrest, Somayaji, and Ackley[13] have proposed an approach which might be called *defense by diversity*, the premise being that widely deployed versions of software are easy to attack because all of their instances are exactly alike. Since a great many attacks are 'canned' (i.e., they are performed entirely by software entities created in advance by an experienced attacker, rather than requiring ongoing human participation in the attack during the execution of an exploit<sup>6</sup>), or are otherwise performed remotely and with limited access (at least until the attack has succeeded), the attack must depend on *a priori*

<sup>5</sup> *Canaries* are named after the canaries and other small birds carried into mines. They have high metabolisms, and would show symptoms of bad air well before the miners: a warning to make a very hasty departure.

<sup>6</sup> The exception is *unwitting* participation (*Trojan horse* malware), as in the "ILOVEYOU" e-mail worm exploiting Microsoft Outlook<sup>TM</sup>: opening an attachment executes the exploit.

knowledge of how the attacked system works: no extensive investigation can be performed during an attack when a surprise is encountered, because either the malware cannot be made sufficiently intelligent, or the cracker does not yet have sufficient access privilege to allow such an investigation. If the *a priori* expectations of the attack's creator or performer can be falsified by diversifying instances of the system, the attack will fail.

Cohen proposes a version of the diversity defense in which application software is modified periodically (i.e., in which the diversity is created *through time*: yesterday's program differs from today's). To implement Cohen's proposal, the system to be protected must be augmented with software which modifies the system on an ongoing basis.

Forrest *et al.* consider diversity in which changes are not successive, but start with the same root software which is then modified in a random fashion (e.g., during compilation). As a result, diversity according to Forrest *et al.* might be termed *spatial* diversity: different system creation instances use differing random input, so that different installations, distributed in space, contain diverse systems.

Whether the diversity is through time as proposed by Cohen, or through space as suggested by Forrest *et al.*, the kinds of diversity which they have proposed are not ideally directed to the specific problems of defending against remote attacks in general and buffer overflows in particular.

For example, Forrest *et al.* propose incrementing stack frame size for routines at compile time, in random increments of 16 bytes, from no increments up to four increments. They found that this did indeed defend against a particular exploit. Suppose, however, that such a defense were used against overwriting the routine's return address, and thus branching to code provided by the exploit. A cracker could simply ensure that her exploit's code entry point is written at *each* of the five possible offsets of the return address, thereby rendering the proposed defense useless. As with canaries, this approach has a *coverage* problem.

We might diversify software using *software obfuscation* techniques[8, 9, 10, 11, 22], which inject some degree of randomness into implementations. However, obfuscation is aimed at confusing an on-site human attacker, rather than foiling a 'canned' or otherwise remote attack: protecting a piece of software against a hostile environment is quite different from protecting an environment against hostile remote accesses or hostile pieces of software.

Our approach uses *tamper-resistant software* (TRS) technology[1, 3, 4, 5, 17, 25, 26] to achieve diversity. Malware operates by tampering with the system; generally by fraudulent invocations or the presentation of fraudulent information. Tamper-resistance to such fraudulent invocations and presentations is exactly what we want.

Our method is to deploy techniques previously discovered in connection with TRS, specialized and fine-tuned for spatially-diversified protection against buffer overflow attacks, and protection against alien executable code. As we will see in the next section, this approach can completely shut down buffer overflow attacks. It also ensures that effective restrictions can be placed on alien executables to prevent them from harming the system.

## 2. OVERVIEW OF THE BABEL DEFENSE

We review here our *Babel defense* (the subject of our recent preliminary patent application[6]). The key concept of this method of defense is to disable fraud by means of communication diversity.

An attempt to perpetrate a fraud generally depends on fraudulent communication. If, by diversifying data representations used in communication, we can render fraudulent communications effectively meaningless, we can prevent the effect which any such fraud was intended to achieve. We call such an approach the *Babel defense*, after the Biblical story of the tower of Babel[24].

The Babel defense protects computer systems against remote attacks by altering the data representations used to communicate information among its components, with corresponding changes to the code manipulating the data in these representations. Different instances of a protected system are created such that representations differ among instances. (If an attack is designed based on *my* instance of the defended system, the attack won't work against *your* instance.)

The Babel defense has the crucial property of defending against *tomorrow's* attack. Unlike virus- and worm-detection approaches, or profile- or rule-driven approaches, it does not depend on historical information. It depends on an ability to invent a variety of dialects by which system components communicate. No cracker can design an attack which anticipates what dialects might be chosen for particular instances of Babel-defended systems or applications.

## 3. THE METHOD: FOILING BUFFER-OVERFLOW AND ALIEN-EXECUTABLE ATTACKS

We describe here the extension of [6] for specific protection against buffer-overflow and alien-code attacks.

We will use the term "**protector**" to denote a program which applies the method of this invention to an application in order to protect it against buffer overflow attacks.

N.B.: We protect the *application* against buffer-overflow attacks, not the *system*. As we will see, the **protector's** user can set the application/system boundary wherever she wishes, within broad limits, so that this is not really a limitation on our method. However, it is important to note that the method protects the parts of the code and data which the **protector's** user defines as *application*, and does not necessarily protect anything outside that boundary. If the user desires broad protection, she must make the boundary broadly inclusive.

**3.1. The Key Observation.** The key observation leading to the current invention is this:

Many kinds of attack, and buffer-overflow attacks in particular, write memory according to one usage, and then employ its contents according to another usage. In the case of buffer-overflow attacks, this is main memory; in the case of alien-code attacks, the memory is represented in a file, and the entire program (when loaded) may itself be an attack region in main memory, normally containing many other regions within the entire program region. (Or a fragment of the program, when loaded, may be an attack region, as in the case of a computer virus, which adds a small amount of hostile code to an existing program.)

Generally, code associates a usage of main memory as a buffer with an array or array area pointer by which it is accessed. (For example, the implementations of most modern programming languages use stack frames to represent the activations of routines, and there is a frame pointer used to access local variables in the frame. This frame pointer is an example of a coarse and imprecise array pointer, which we might use if more detailed information on arrays were not available, e.g., in protecting a program in binary form. Another example is the area pointer returned from the `malloc()` routine of the C standard library, which points to a dynamically allocated and freed block of memory which is implicitly an array.)

If we can prevent memory written according to one usage from being used effectively by a cracker or by malware for some other purpose, we can foil any attack which depends on such multiple usages. That is, if we can use main memory encodings associated with specific arrays or approximations thereof, then when data addressed using one area pointer overflows into another area, the overflow data is wrongly encoded for the memory into which it overflows. Hence the overflow data is nonsense in the new area, and cannot have the effect intended by the cracker.

That is, the objective is to encode data so that overflows become *meaningless*.

**3.2. Making Buffer Overflows Meaningless.** We consider how to apply encoding to individual arrays (or approximations thereof, when precise information is not available and we have to fall back on information obtained from binary code by decompiling, for example).

**3.2.1. Partitioning Code and Data Memory.** We divide code into:

- *Application code*: Code which is part of the particular application and is expressed in its source code.
- *System code*: Code which is part of the run-time support for the application code, consisting of *library code* for basic utilities supporting allocation and deallocation of memory, input and output, and the like, or which is part of the internal representation of abstractions provided by the library code, or which is automatically generated or included by the compiler, linker, or loader to support the abstractions provided by the language in which the source code is written.

The division of code into application and system code is static in many contexts.

We divide data memory into:

- *Application data*: Parts of memory which are accessed by code expressed in source form in the *application code* during normal operation; i.e., within the normal assumed behavior on which the code, the compiler, the linker, the loader, and the library are based.
- *System data*: Parts of memory which are not application areas, and which are or may be accessed by the library code, or code which is automatically generated or included by the compiler, linker, or loader to support the abstractions provided by the language in which the source code is written.

The division of data memory into application and system data varies dynamically in many contexts. For example, when the system allocates a block of memory for use by an application, the portion of the allocated block which is intended for use by the application (as opposed to any system information about the block kept just beyond the limits of the application-usable part) changes from being system memory to application memory. When a routine is called, system code generated

by the compiler allocates an *activation record*<sup>7</sup> so that the local variables in the activation record change from system data memory into application data. The reverse of these changes occurs when an application calls a system API to return a block of allocated store to the system, or when a routine returns from a call, freeing its activation record.

N.B.: The division into system and application can be done at various levels. For example, for the operating system, the language support and the kernel code are the system, and the operating system itself is the application, but for a program running on top of the operating system, the operating system is now part of the system and only the code of the program, minus its library support, is application code. A similar consideration applies to the data.

3.2.2. *What is a Buffer?* A buffer is a contiguous region of data memory into which an incoming message, argument, or other communicated value, generally in the form of a sequence (typically, a sequence of characters), which is passing from the external world *via* the services of system code, into data memory, or from which an outgoing message, parameter, or other communicated value is passed to the external world *via* system code services. For our purposes, we are interested in *application buffers*, which are buffers residing in application data memory.

We note that the form of buffers in a language such as C is somewhat stereotyped. A buffer is generally one of the following:

- *Explicit array:* An explicitly declared array (typically, an array of characters).
- *Implicit array:* A contiguous region of allocated memory with a pointer initially pointing to its lowest (or more rarely, to just below the lowest, or to its highest, or to just beyond its highest) application data address (typically, a character pointer). The type of the pointer target is the type of the elements of the implicit array.

In code generated from C, and often in code generated from C++, the pointer does not provide bounds for the array (unlike the case with languages such as Algol 68 or Protel).

Note that, whether explicit or implicit, the actual array (as opposed to its type, if any) actually has a specific size. The system code has allocated some fixed amount of space for it, and any use of memory beyond that allocation is necessarily an overflow.

In a language such as C, we do not find scalar values or structured values created using `struct` or `union` constructors being used as buffers. Therefore, if we are expecting overflows, our attention must be focussed on arrays. In C++ or other languages, there are further possibilities, such as use of the `vector<>` template, but the basic considerations remain the same: buffers are generally strictly contiguous, sequential, flat structures in which elements are added successively at increasing (or, more rarely, decreasing) physical addresses, having a fixed upper (or, more rarely, lower) bound beyond which any further addition is an overflow<sup>8</sup>.

<sup>7</sup>The record containing the local variables of the routine, which are application data, together with system data indicating the routine address, the location of the activation record of the calling routine, etc.

<sup>8</sup>In fact, the other possibilities are implemented by underlying explicit or implicit arrays.

3.2.3. *What to Encode and How to Encode It.* We strive to encode every implicit or explicit array which might be a buffer, especially if it might be an input buffer. We strive to encode distinct encoded arrays differently. We say 'strive' to encode, because if we are encoding a program in binary form, the division of code into arrays and scalars, and into system and application, are likely to be less complete and precise than in the case of a program for which source code is available. As a result, we must fall back on imprecise methods of estimation, well known in the art of decompilation, which will generally give a reasonably close approximation to these divisions, but which will not be accurate in every case. Nevertheless, the estimates will be good enough to make our defense quite effective, if not absolutely perfect.

Although some protection is provided when all encoded arrays use the same encoding (specifically, it protects against attacks involving overflow from application data into system data or into code), we *prefer* also strive to encode distinct encoded arrays with distinct encodings. We say 'strive' to encode distinctly because

1. with a binary program, we may not know as precisely as we would for a source program where all arrays begin and end, and
2. depending on our encodings, there may not be enough distinct encodings to guarantee that we can ensure that encodings of arrays are unique for each encoded array.

But whether or not we are able to make the encodings of distinct arrays distinct, we *prefer* to make the probability that encodings of two distinct arrays match small. (Note that the probability that the encoding of an encoded array matches that of system data is zero, since system data is not encoded.) For example, if we have 255 distinct encodings, the probability for an arbitrary pair of encoded arrays is less than 0.4%, which means that attacks which work on unprotected applications by overflowing from one buffer into another have a low probability of success on applications protected according to this invention. Of course, a determined cracker could keep trying encodings until a hit was discovered, but even for purely byte-wise encodings, there are thousands of reasonably fast encodings from which to choose. If we use encodings with more choices (e.g., some reasonably fast encodings provide billions of choices) we can protect applications against even concerted brute-force attacks.

Since we are defending against remote attacks, we do not need much sophistication in the encodings themselves. Indeed, it is highly preferable that we keep them reasonably simple, since it is far better if the **protector** can be made so very reliable that we can count on it to work correctly every time it is applied to an application — a principle which has been uniformly applied in the preferred embodiments which follow.

We prefer that encodings be chosen to have the following characteristics:

1. They should not change the size of data items, since such a change would incur substantial complications in processing programs to install our defense, especially in the binary code context. Such a change is also likely to result in a substantial execution overhead.

Whether this is merely a preference or a requirement depends on the efficacy of program analysis to determine the types and bounds of arrays. The more accurate the analysis, the more freedom we have to change the sizes of

data items. However, it is certainly simpler to handle the case where data sizes are not changed.

2. They should be performable using inline non-branching, non-conditional code in a small number of instructions, since this is what will provide the best performance on modern computer hardware.

This is not required, but it is advisable for performance reasons.

3. They should not change the sizes or addresses of data memory fetches or stores, since such changes might also incur high execution overhead.

Again, if the analysis of the code is very effective, this is optional. However, it is easier to avoid the complications incurred by changing data sizes when we protect an application.

Examples of such encodings are:

1. The encoding and decoding are done by a bit-wise exclusive-or of each byte,  $\vec{B}$ , of the data with a non-zero byte constant, say  $\vec{K}$ . I.e., the encoding/decoding formula is  $\vec{B} \oplus \vec{K}$ . We have used the vector-arrows, as in  $\vec{B}$  and  $\vec{K}$ , to emphasize that we view these quantities as bit-vectors in the computation, and that the  $\oplus$  operation is a bit-vector exclusive-or.

(For any size of data from a byte up to the largest size of data for which an exclusive-or operation is provided in the hardware, this permits the encoding or decoding to be done by a single hardware operation.)

This provides 255 distinct possible encodings.

2. As in the above encoding, but encoding/decoding uses the formula  $\vec{B} \oplus \vec{K} \oplus \vec{A}$ , where  $\vec{A}$  comprises the low-order eight bits of the address of the memory byte from which  $\vec{B}$  is fetched or into which it is to be stored.

This provides 255 distinct possible encodings.

3. The encoding and decoding are done using an exclusive-or of each byte of the data such that aligned byte-quadruples (typically, *words*) of the data are exclusive-or-ed with a non-zero byte-quadruple value. To put it another way, there are four byte constants used, and which is applied to a given byte in an encoded array depends on its position relative to the beginning of the aligned byte-quadruple containing it. The encoding byte constants are  $\vec{K}_0, \vec{K}_1, \vec{K}_2, \vec{K}_3$ , and the encoding/decoding formula is  $\vec{B} \oplus \vec{K}_{A \bmod 4}$ , where  $A$  is the memory address of the byte. (Note that the modulus can be computed by a bit-wise and operation, so the required additional machine code is compact and very fast.)

This is more complicated, since the low-order two bits of a byte's address become relevant, but it provides an enormously larger variety of encodings:  $2^{32} - 1$  of them — over 4 billion — instead of 255 of them.

4. As in the above, but using 16-byte groups instead of byte-quadruples. The constants are then  $\vec{K}_0, \vec{K}_1, \dots, \vec{K}_{15}$ , and the encoding/decoding formula is  $\vec{B} \oplus \vec{K}_{A \bmod 16}$ , where  $A$  is the address of the byte. (Again, the modulus is compact and very fast using bit-wise and.)

This is slightly more complex than the above encoding, with the low-order four bits of the address relevant to the encoding and with sixteen constants to use for each coding, but it provides  $2^{128} - 1 \doteq 3.4 \times 10^{38}$  possible encodings. For such an encoding, a brute-force remote attack or attack by viruses or worm programs is like a brute-force attack on 128-bit AES encryption: it is totally infeasible in practical terms. (After the attempted attack has been

proceeding for several years, someone is likely to notice and take steps to prevent its continuance.)

5. Encoding is done by adding a byte constant  $K$  (modulo 256) to each byte, and decoding is done by adding the complementary constant  $256 - K$  (modulo 256) to each byte.

This provides 255 distinct possible encodings.

6. Encoding is done as above, and then performing one of the possible exclusive-or-based codings, and for decoding, applying the decodings in reverse order.

This provides over 32 thousand possible encodings.

7. To encode a word, add some constant  $K$ , modulo the natural word modulus  $W$  (typically,  $W = 2^{32}$ ), using unsigned (or equivalently, 2's complement) arithmetic with overflows ignored; to decode a word, add the constant  $(W - K)$ , modulo  $W$ , using unsigned (or 2's complement) overflow-ignored arithmetic. Encoding or decoding can then typically be done in one hardware instruction, just as with exclusive-or, since the modulus operation is provided by the normal hardware behavior of addition with overflows ignored.

This provides  $2^{32} - 1$  possible encodings.

Many other codings will be derivable by compiler experts from the teachings herein.

In general, we can use any *bijective* encoding<sup>9</sup> which is not the *identity* encoding<sup>10</sup> and which permits us to encode data of every size used in connection with the encoded array. We prefer, however, that the encoding be implementable by inserting straight-line code, with at most a few instructions to encode, and a few to decode, and with a high execution speed. Whether we must handle small sizes, such as a byte, or only a fixed size, such as double-bytes or byte-quadruples, depends on the type and usage of the array. If data is always accessed in larger sizes, the number of convenient, distinct possible encodings is larger. As noted above, by using address-dependent codings, such as the byte-quadruple or 16-byte encodings above, we can — at some increased cost in overhead — lock out arbitrary brute-force attacks, no matter how concerted.

Even if we just use an 8-bit exclusive-or mask, the probability that a given piece of malware will succeed on a random Babel-protected instance is only 0.4%, which is low enough to slow down the propagation of self-replicating malware, such as a virus or worm program, enormously. Using only the byte-wise encodings listed above, there are over 32 thousand choices, which would give a probability of about 0.003% of success, and would take an average of over 16 thousand tries to crack using a brute-force attack. Considering the number of steps and time it takes to perform such an attack remotely, and the time it takes to execute such attacks remotely, this is already sufficient to lock out all but the most extremely patient crackers, or crackers using heavy-duty attack automation.

If we want a well-nigh perfect defense against programmed attacks in general, we can use some encoding such as the 16-byte-group encoding which raises the average number of tries needed for penetration by brute-force searching totally beyond what is currently feasible using a remote attack.

Note that the brute-force attack is the *only* feasible remote attack on any given buffer-overflow vulnerability, because getting more refined information other than success or failure takes a high level of privilege, which normally cannot be obtained

<sup>9</sup>A *bijective* encoding is one which does not lose information.

<sup>10</sup>The *identity* encoding leaves the data unchanged.

until *after* the attack succeeds. (Of course, the system under attack may have non-buffer-overflow vulnerabilities, so that penetrating the buffer overflow defense is not necessary: see 3.2.5 on p. 13.)

3.2.4. *Why We Encode: Effect on Buffer-Overflow Attacks.* Here, we describe the general effects of applying the above methods on buffer-overflow attacks.

*Stack Overflow Code-Execution Attack:* Suppose an application has the following vulnerability: there is a message buffer stored in a local variable declared in a routine, which will be allocated on the stack at run time. (For example, this kind of vulnerability was found in the Unix<sup>TM</sup> `rdist` utility for Solaris<sup>TM</sup> 2.\* and SunOS<sup>TM</sup> 4.1.\*, and for other systems running an old `rdist` with `setuid root` selected[19].) Suppose we are running on the (most common) kind of processor, where stacks grow towards lower addresses, with a routine's local variables stored below the linkage, so that the buffer overflow can overwrite the linkage information for the routine, including the return address.

A cracker engineers a situation which results in an overly-large message being placed in this buffer. The return address is overwritten with the address of the next word, and the remainder of the message consists of executable machine code to do whatever the cracker wants, using the privileges of the application itself — say, propagating itself over the Internet using the user's address book, and then wiping the computer's hard drive.

When the application receives this message, it is placed in the buffer, overwriting the return address with the address of the next word, which begins our desired machine code constituting the remainder of the message. When the routine attempts to return, it branches to the code location now stored in the return address, resulting in the execution of the machine code placed at the end of the message, as if it were the application's own code, propagating itself over the Internet by means of the victim's address book, and then wiping the computer's hard drive.

Now consider what happens when this attack is applied to an instance of the application protected according to this invention. The message is placed in the buffer (in the encoded form for that buffer, since it is ostensibly data), and it overwrites the return address, as expected. The application sees the message just as the undefended application did, because data items are encoded on write and decoded on read. However, when the routine attempts to return, the return address is *not* stored as the address of the next word, but rather some random address resulting from its encoding, and the remainder of the message is *not* the cracker's desired machine code, but rather some strange string of bytes resulting from its encoding. As a result, when the return address (which is not decoded on read, since it is in system data memory) is used for the return, the code branches to some wild location, and the application crashes.

Crashing may not sound ideal, but

- it is certainly less painful than losing the contents of the hard drive and taking down the local intranet, not to mention a number of colleagues elsewhere, and
- it is exactly the kind of behavior the unprotected application would have shown, when handed an overly long message which was *not* part of an attack.

Suppose the cracker obtains access to a particular instance of the application protected according to this invention. She can then determine the encoding for the particular buffer and adjust the attack's message so that it allows for the encoding

used in *that particular instance* of the application. However, it is useless for attacking *any other* instance, since the encoding will be different and the adjustment won't work.

Hence, using this invention, we defend against *outsider* attacks by using a different variant from companies or organizations. We can also protect against *insider* attacks by refreshing the instances of applications, using varying encodings, at random intervals, so that specialized insider knowledge of particular buffer encodings gained today is useless tomorrow.

*Static/Heap/Per-Process Overflow Scripting or URL Attacks:* The effect of the Babel defense on other forms of attacks is similar.

Let us consider other categories of attack. Communication buffers are located in three kinds of memory: the routine-activation *stack*, which holds the run time data of routines during their execution, *static* memory which is part of the program as loaded, and *heap* (dynamically allocated) memory which is allocated while the program is running by some utility such as `malloc()`. There may also be data specific to a particular thread of control, a concept supported in Microsoft Visual C<sup>TM</sup>.

Buffer overflow attacks generally seek to make something illicit happen, either by causing machine code provided by the attack to be executed, or by passing a message which causes the attacked system or application to perform an illicit action (e.g., by providing a command script or an URL).

Consider the scripting or URL possibilities. Suppose the buffer resides in the heap or in static memory. The cracker may either make the message overwrite an array where a script or an URL was expected by the application, or it may pass both a script or URL address where the application expected a script or URL address, such that the address points to a part of the message containing the illicit script or URL. The script can take any hostile action permitted by the scripting language; the URL can act as a gateway for other hostile scripts or other web-based attacks.

In any case, when the cracker's message is placed in the buffer of the intended victim version of the attacked application (or system), the overwriting intended by the cracker occurs, resulting in the eventual execution of the hostile script or use of the hostile URL, and the attack succeeds.

However, when the application is protected using the methods of this invention, the following behavior ensues (recall that various buffers are generally encoded differently):

Either the hostile script or URL overflows from one buffer into another array, so that when the application attempts to execute the script or use the URL, it fails because it was written with the encoding of the overflowing buffer, but read (with high probability) with the *different* encoding (or, if the array is known not to be a buffer, no encoding) of the array into which it overflowed, or the address was written as data, and hence encoded with the encoding of the overflowing buffer, but either read with no encoding (because it is not in a buffer) or read with the different encoding of the array in which it resides, so that the address is wild and does not point to the illicit script or URL as the cracker intended.

Various results may follow, ranging from an 'illegal script' or 'illegal URL' error to a crash. Again, the damage intended by the cracker is avoided, and the possible responses of the application are exactly the same as for overly-long messages which are *not* part of an attack.

And again, access to a given instance protected according to this invention does not provide a means of attacking other instances, so that this invention can be used to defend against both outsider and insider attacks as described in 3.2.4 on p. 11.

**3.2.5. Can the Encoding Defense be Penetrated?** There are ways in which the encoding defense according to this invention can be bypassed or penetrated:

*Configuration and Protocol Flaws:* There certainly may be ways to penetrate the defense provided by this invention using flaws in protocol design or incorrect system configuration (e.g., leaving an exposed guess-able password providing privileged system access over the Internet).

This invention protects against buffer overflows. It cannot protect a system whose normal defenses have been weakened by improper configuration or programmed-in permissive protocols, which permit the defense to be, not penetrated, but bypassed entirely.

*Insufficient Knowledge of the Protected Application:* The other major kind of penetration which is possible occurs when the invention is applied with insufficient information about where arrays begin and end, making perfect coverage impossible.

This can happen when the invention is applied to binary code, for example: decompiling is inexact in the current state of the art. This may result in multiple arrays being treated as single arrays, or entire routine activation records being treated as single arrays, which might permit some exploits to occur by making encodings common where, were the available information about the code greater, they would be encoded distinctly, so that the defense against an overflow from one into the other is omitted.

This invention depends on information about arrays, and especially arrays used as buffers. We can always make the assumption that any array which *might possibly* be a buffer is in fact a buffer (thereby encoding more arrays than strictly necessary, with higher run-time overhead but no weakening of protection). However, we cannot always guarantee to handle situations where arrays cannot all be separately identified, which may result in an area of memory which, according to the source code, contains multiple arrays, as if it comprised a single array. As a result, some overflows may be defended only by a common encoding where multiple encodings would be more effective.

As a result, certain buffer overflow attacks may still be effective. (Those which depend on compromising system data, or data outside the arrays discovered during the protection process, will still fail, but those which depend on overflow from one application array into another application array, may sometimes succeed.)

*Foiling Buffer-Overflow Attacks with Good Information:* Penetrating the defense provided by this invention for code in *source form*, or other code which can be *effectively analyzed* by whatever means, using a buffer overflow attack, can be made very difficult indeed, and (with full deployment) well-nigh absolutely infeasible.

The reason is simple: a buffer overflow attack can only succeed when it can initiate its first illicit action. This invention's defense, when deployed with full coverage, converts the illicit action to nonsense behavior, so the first illicit action, with high to extremely high probability, cannot occur. Hence any subsequent steps which might have happened are irrelevant: the attack won't even make it to its first step.

Even relatively simple encodings, such as a 32-bit exclusive-or mask applied to each 32-bit word (quad-byte encoding), provide massive diversity, and hence very low probability of attack success: the cracker has one chance in about four billion of guessing right. (If the attacker sends millions of exploit instances looking for the right mask, someone is likely to notice.) 128-bit masks (16-byte-group encodings) make the search space not only impractically large, but infeasibly large: attempting a concerted crack would be like a brute-force search for an AES cryptographic key, but with very slow steps (it takes time for malware to make its way over a communication network).

The only way a cracker can penetrate this invention's defense, when the full power of the defense is deployed using comprehensive information about the protected code, is to have access to the specific attacked instance when designing the attack. But this has very limited payoff for the cracker because all other instances are different, and the instances can be varied over time as well.

*Foiling Most Buffer-Overflow Attacks with Incomplete Information:* Since decompilation is fairly effective, we can expect to provide *good* protection for binary code, but since it is not perfect, we cannot expect to provide *perfect* protection for binary code.

Even when the information available is incomplete, the protection provided by this invention can make a successful attack *much less probable*. That is, even if the defense is applied to binary code and is not as comprehensive as for source (unless sufficient auxiliary information is available to fill in the required characteristics of the code such as exactly where arrays are), or even if we intentionally use lower levels of diversity (simpler, faster, more compact encodings) for the sake of maximizing execution speed, deployment of the invention has strong economic justification:

- It makes it *much less likely* that a given attack will succeed, whether the attack is a buffer overflow attack or an alien executable attack (see 3.3.3 on p. 21).
- It *slows down tremendously* the rate of propagation of buffer-overflow-based self-replicating viruses and worm programs over the Internet.

That is, deployment of the invention's defense constitutes good Internet citizenship: the user makes the Internet safer for other 'netizens', as well as herself.

- Therefore, it will *significantly reduce the costs* induced by remote attacks over the Internet. (Since buffer-overflow vulnerabilities comprise many of the most serious vulnerabilities, they account for a high proportion of the economic damage caused by remote attacks. Alien executable attacks — see 3.3.3 on p. 21 — are another major remote attack category against which the invention also provides protection.)

**3.3. Other Aspects of Protection.** Up to this point, we have only considered the *interior* of an application; i.e., the application minus any input from the outside world or output to the outside world. We have also assumed that, given the encodings for regions, the program which transforms an application to its protected form can always determine which encoding is appropriate for any pointer into that region.

We must now consider the application, including *pointers with non-static codings*<sup>11</sup> and *in its containing context*: we must deal with

- the transition from the exterior of the protected application to its interior and *vice versa*;
- the cases where a pointer's coding must be determined dynamically; and
- applications which may have been compromised by the attachment of *virus* code, or applications such as *worm programs* which may themselves constitute an attack.

3.3.1. *Handling Application Boundaries*. Recall the division of code and data into system and application parts in section 3.2.1 on p. 6. For our present purpose, the application boundary is crossed by data being passed by the system code from system memory or an input device into an application buffer, or being passed by system code from an application buffer into system memory or to an output device. The boundary lies between information managed by the system and the data arrays which are part of the application.

Up to this point, we have simply assumed that any data crossing this boundary was properly encoded for its receiving location, except that it might possibly occupy too much space and thereby overflow into data beyond its normal extent.

We now describe how this boundary-crossing encoding is actually provided in connection with our defense.

*Identifying Input and Output Routines*: Input and output are generally handled by specific routines provided by the system, which are called with the application buffer as an argument in order to perform the desired input of external information into the application buffer or output information from the application buffer into the external world. This system routine may move the data to or from a hardware device, or to or from intermediate memory and thence subsequently to a hardware device, in various ways. The information may also be transferred from one application into another application, as when a message is passed from one running process in the system to another. Whatever the handling of the information by the system, from a given application's point of view, it is generally either an input or an output, or (in some cases) both.

For our purposes, if any such input and/or output routine is converted to *inline* code embedded in the application code, we consider it to remain part of the system if it deals with system data, and part of the application if it deals only with application data.

A special case of input followed by (or intermixed with) output occurs when a system routine is used to transfer data from one location to another or to process a block of application data and then either modify it or return a sizable result in another location. For example, in the standard C library, routines such as `strcpy()`, `strncpy()`, and `memcpy()` have such behavior.

These routines are very easily identified when source code is available, of course, because their identification is clearly specified by system libraries. Even in the binary context, because of the stereotypy of their code or their association with specific code locations (such as in connection with shared objects in Unix<sup>TM</sup> or

---

<sup>11</sup>I.e., pointers for which the protector was not able to determine statically which coding to use, so that a dynamic coding determination must be used in the running application.

GNU/Linux systems, or dynamic link libraries (DLLs) in Windows<sup>TM</sup> systems), identifying such routines is not difficult.

The first step in handling the application boundary, then, is the identification of all calls to such routines and the arguments and results of these calls.

*Filtering Input and Output Routines:* After we have identified all calls to input/output routines (see p. 15), we must *filter* their behavior by either changing the code to use a variant routine provided as an extension to the library, or adding code to pre- and/or post-process the results of the call.

For example, consider the standard C library function `strcpy()`. It copies a null-terminated string of characters (normally bytes) from one place to another within the application's memory. The *from*-string is expected to be terminated by a null (character code zero) character. If the string is encoded, though, the final character of the string, which is read as zero by protected code (which decodes it), is actually stored according to the encoding: i.e., it is actually *not* zero.

If we use `strcpy()` 'as is', and with no restrictions on our codings, the copy may continue beyond our string to copying the following data until it eventually encounters a null character or runs entirely out of addressable memory, or it may stop copying before the entire string is moved because it encounters a character whose non-zero character code happens to be *encoded* as zero in memory. This simply won't do.

To fix this, we do one of the following (or a variation thereof: such variations will be derivable by a compiler writer from the teachings herein):

1. Provide an additional library routine — say, `strcpy_encoded()` — which takes a description of the encoding and decoding in effect for the *from*- and *to*-strings and uses these to read the characters, including the terminating sentinel character, in the *from*-string according to its encoding, and to write the characters, including the terminating sentinel character, in the *to*-string according to its encoding.

The **protector** then changes `strcpy()` calls into `strcpy_encoded()` calls wherever an encoded *from*-string or *to*-string (or both) is involved.

The additional library routine can either be written directly into the application code by the **protector**, or we can actually add a separate new routine to the library used when linking together application code.

Since this provides the most opportunities for optimization, it is likely to provide the best performance.

2. The **protector** changes the code so that, just prior to the call to the `strcpy()` call, the *from*-string is decoded according to its encoding, and just after the `strcpy()` call, the *to*-string is encoded according to its encoding.
3. As in the previous approach, but we add a library routine — say `str_recode()` — which performs the service of recoding (i.e., decoding input, encoding output, or combining the decoding of input and the encoding of output into a single combined recoding as characters are transferred) strings according to a description of the recoding, and performing the encodings and decodings of the previous approach by inserting appropriate calls to this routine into the application code so that the **protector** can ensure that they are performed just before and just after the call to `strcpy()`.

(All of the above provide the same level of protection: the only difference is in performance.)

Approaches such as those above, and their variants, suffice to handle *all* data crossing the application boundary from the exterior world into application buffers or from application buffers into the exterior world. In particular, I/O routines are handled by simplifications of the above approaches (or variants thereof), since they are normally unidirectional, whereas `strcpy()` is bidirectional (both input and output).

3.3.2. *Identifying Codings at Run-Time.* In the foregoing, we have assumed that coding is statically selected: i.e., except inside routines such as `strcpy_encoded()` or `str_recode()` recently mentioned in the previous section, the **protector** can always determine statically what coding, if any, is appropriate for memory accesses with any given pointer into a buffer or potential buffer. (Note that the coding may vary over time, since a given pointer may point into more than one buffer during the execution of a given application.) We must, in other words, disambiguate pointer usages. This can be done statically by alias analysis or dynamically by various other means.

This statically analyzable case is certainly a common one, and the **protector** should be designed to take full advantage of this fact for the sake of performance. However, it is far from universal.

The general aliasing problem for sufficiently powerful programming languages, such as C, C++, and their allies, is *undecidable*; i.e., it is known on theoretical grounds that it cannot be solved statically in general. More restricted aliasing problems, which involve restricting assumptions (or require restricted contexts) either provide *limited information* or are *computationally prohibitive*.

This being so, there will inevitably be circumstances in at least some applications where the **protector** must add to the protected application some means for dynamically identifying the appropriate coding for memory accesses *via* a given pointer at run-time (i.e., for disambiguating pointer usages at run-time).

In addition, there are circumstances in which it is desirable to provide varying encodings for pointers handled by the same code. For example, in a group of one or more routines which are part of an inter-recursive set (i.e., which lead to call graphs containing loops), there is a possibility that a buffer in one of these routines in one activation record overflows into a corresponding buffer of the same routine in a different activation record. To protect against buffer overflow attacks making use of this, the two buffers must have good probability of being encoded differently, which requires that their encodings be chosen dynamically.

Here we describe ways in which such dynamic association of codings with pointers can be achieved.

*Fat Pointers:* Conceptually the simplest way to handle the coding identification problem is for the **protector** to change the format of a pointer to indicate which coding applies to the data to which it points. We call this the *fat pointer* approach because the pointer contains additional information, and may, in fact, be physically larger to make room for it.

This is certainly feasible for some applications, particularly where we have the application in source form (or in binary form but with a good deal of auxiliary information). However, it requires that the *protector* add filtering code for pointer values as they are transmitted between the system and the application, since the system will not handle such pointers in general, unless we propose to provide alternatives for every routine interface in the system dealing with pointers — a major

undertaking in itself for most systems. That is, the pointers must change from fat to ordinary when crossing into an unmodified system routine, and from ordinary to fat on the way back. (And converting from normal to fat may require another technique, such as the *mapped memory* approach described below).

In addition, if fat pointers are *physically* larger than normal ones (i.e., if we can't fit both a coding identifier and the address into the space normally used for a pointer), then we need good analysis to make the use of fat pointers feasible for an entire application in the first place.

Moreover, in applications where many multi-linked dynamically allocated structures are used, pointers may occupy a significant portion of the allocated space, and hence fat pointers may make a significant difference to a protected application's memory requirements.

Also, even when the coding for the target data of a pointer must be determined dynamically on occasion, it would often be possible to determine statically that two accesses, whether from the same or different pointers have the *same* coding (e.g., one of the pointers might be derived from the other in a manner which guarantees this). Therefore only *some* of the uses of a given fat pointer would require extraction of its coding information.

However, the fat pointer approach may be eminently suitable for handling the dynamic coding identification problem for *some* of the pointers in an application, even in programs where it cannot or should not be used to solve the problem for *all* of the pointers.

*Mapped Memory:* Another approach is for the **protector** to modify the application so that it keeps some form of mapping information which indicates, for a given address-range in application data memory, what coding applies to it. (Such a mapping need not be applied for *all* of the encoded arrays in application data memory, but only to those which may be or may contain pointer targets, where the pointer cannot always be associated with an appropriate target coding statically or by use of the *fat pointer* approach described above.)

Let us call this mapping information the *code map*. Each entry in the map contains data providing the beginning and ending of a region of memory and a code identifier indicating which coding should be applied for it. (Regions of memory which are not coded can be omitted from the map.)

This has the advantage that the format of pointers is not changed, so no filtering is required when pointers are transferred between the application and the system. It has the disadvantage that the code map must be searched to determine the coding to be used for the target data for a given pointer, which is slower than simply extracting the information from a fat pointer. (On the other hand, other pointer accesses which are determined to use the *same* encoding as the one determined by a particular reference to the code map are potentially faster, since there is no overhead to handle the extra information or the non-standard pointer format which is used for a fat pointer.)

N.B.: If *every* access *via* a pointer uses the information in the map, then overflow information will be encoded *correctly* instead of *incorrectly* when a buffer overflows. For this reason, we extend the approach so that, this will almost never occur in practice, because of the way we will use the map. The key point is that buffer overflows tend to happen in one access, or in a sequence of accesses, using either a single pointer passed to the system, or a series of successive pointers which can

easily be determined to be derived from their immediate predecessors. In such cases, we require that the code map be used *initially*, and that the other codings simply be *determined statically to be the same as the initial one*, so that the overflow will indeed be encoded *incorrectly* for the region into which it overflows, which is the property on which the encoding defense depends. Not only does this ensure the desired protection, but it improves performance.

There are many ways in which such a memory map can be constructed which are known in the art of main-memory data structure design. Hash tables are a good example. One particular variant of the hash table approach is to use a truncated address, in the manner of a hardware direct-mapped cache memory.

Our preference, however, is that the map be stored as an AVL tree[16], which has the advantage that entries can very easily be added to or removed from the code map as potential buffers are allocated or freed. It also has the advantage that pseudo-entries for *uncoded* regions of memory are easily computed using the code map, which is useful in connection with the caching method described below.

Uses of the code map will occur during the execution of the application, whenever a pointer is used for which the protector was not able to determine statically which coding to apply. This can happen either because the memory was dynamically allocated in some group of allocations which have no sufficiently specific static association, or because it cannot be determined statically to which of the potential buffers with known codings the pointer will point at run-time.

The storage for the map can be allocated separately from the other application data memory, but this has the disadvantage that maintaining it as it may grow and shrink during execution may incur added overhead.

We therefore prefer that the map be distributed in association with the pointer usages as follows:

1. If a potential buffer which needs to be represented in the code map lies within static storage allocated when the application is initially loaded, the PROTECTOR should augment the initial allocations for the application with sufficient added space to store an entry of the map (e.g., a node of the AVL tree) for that potential buffer. The loaded program, when it is loaded, can then contain an initial version of the map which contains these entries (e.g., these nodes of the AVL tree), which are always present: they are never removed from the map.
2. If a potential buffer which needs to be represented in the code map lies within a routine stack frame, the PROTECTOR should augment the stack frame with added sufficient added space to store an entry of the map (e.g., a node of the AVL tree) in the frame for that potential buffer. It then adds code to add such nodes to the map when the routine is entered, and to remove them from the map on exit from the routine.
3. If a potential buffer which needs to be represented in the code map lies within a block of dynamic store allocated by a call to an allocator routine such as `malloc()`, the PROTECTOR should augment the allocation with added sufficient added space to store an entry of the map (e.g., a node of the AVL tree) in the frame for that potential buffer. It then adds code to add such nodes to the map when the block of storage is allocated, and to remove them from the map when it is freed (e.g., by a corresponding call to the `free()` routine).

**Encoding Caches:** If the code map described above were used naively, each access to the map would be much slower than a normal pointer access. To increase the

efficiency of the above approach, we prefer that the **protector** insert a *code caching* strategy into the protected application, as follows:

One of the very common cases is that accesses to memory made by a particular instruction sequence in the code (which sequence may be executed many times during the run of the application) will often use the same mapping on two successive executions, or will use a small number of mappings within a group of successive executions.

We can take advantage of this 'coding locality' in much the same way that a data cache takes advantage of localities in data addresses. We store the mapping information in locations allocated for that code, either in static globally allocated memory, or in the stack frame for the routine containing the code sequence in question. We can store the mapping for the most recent access, or the last  $n$  mappings used, where  $n$  should not be too large — say, not more than five entries. Then whenever locality works to the application's advantage, a slower lookup in the code map is replaced by a faster lookup in this cache. (Note that an entry in the cache may either be an entry from the code map, or a pseudo-entry indicating that a region of memory is not encoded.) Let us call each of these small caches an *encoding cache*.

In particular, if the number of entries is small, the lookup code for the cache can be inline instructions rather than a routine call, and the lookup can be quite fast whenever the cache contains coding information for an address range containing the pointer target.

Note also that, wherever we can determine that the coding for a particular pointer is the same as that for a previous access whose coding is either statically known or was derived from one of the above-mentioned caches or from the code map, no lookup in either one of the caches or the code map is required, and the access occurs with no overhead beyond the encoding or decoding itself, if any.

**Combined Approaches:** As noted above, the *fat pointer*, *code map*, and *encoding cache* approaches described above can be variously implemented and variously combined to optimize the efficiency of the needed dynamic identification of data codings. In fact, we prefer that such combinations be deployed wherever they can improve the efficiency of the code produced by the **protector** for a given application.

Many variants of the above techniques would be derivable by experienced compiler writers from the teachings herein.

**Passing Data To and From the System Using Run-Time Code Information:** It often happens that system I/O routines are provided with access to application data by pointers which are passed to the system routine. One approach to handling this is to modify the system routines along the lines previously discussed in connection with the application boundary in the section on filtering (see p. 16).

When this is done in connection with *fat pointers* or pointers for which the appropriate coding must be derived using the *code map* or *encoding caches*, we prefer to handle this situation by decoding the data immediately prior to calling the system routine, and encoding the data on return from the system routine. Thus the system routine needs no modification, and the encoding needed for the defense is preserved whenever the data accessed using the pointers are under the control of the application.

That is, we basically treat passing the data to the system as we would a *read*, and receiving any data back from the system as we would a *write*, of the encoded region of memory.

**3.3.3. Loaders and Interpreters: Handling Alien Executable Code.** Executable code from external sources may arrive in the form of a so-called *executable file* containing specially formatted machine code and data, or in the form of a *script file*, containing instructions which must be interpreted in some way before they can be executed by a computer.

Both executable files and script files can be, and often are, transmitted over the Internet, thereby becoming *alien* executables, and both can contain a cracker's hostile commands: i.e., both can be harmful — and sometimes *extremely* harmful — on a computer which runs them.

**Loaders Execute Machine Executables:** Generally, an *executable file* (let us call it a *machine executable*) contains data in a special format which is intended for loading and execution by a *loader* (a program which converts the data in the file into executable machine code in main memory, and transfers control from itself to the starting location in this machine code).

The binary representation of machine instructions and data in such a file is normally very close to their binary representation in main memory, but certain blocks of data memory which are pre-allocated when the loaded program begins to execute (BSS blocks and the like), are not represented by blocks of data in the file, but are represented by an indication of the block's address, size, and possibly initial data contents (such as all zeros), and certain addresses and identifying numbers or codes (*fix-ups*, such as the addresses and indices of routines within shared objects or DLLs) are not represented in the file, but are provided in some encoded form which is replaced by the appropriate actual values, numbers, or codes as the code is loaded.

**Interpreters Execute Scripts:** In the case of a *script file* (let us just call it a *script*), the format of the executable code is typically quite different from the machine code of the computer on which it is to be executed. Instead, the code is executed by an *interpreter* of some kind. The interpreter may operate in various ways:

1. It may simply read the code and obey the encoded commands in it one at a time, so that the number of times a command is interpreted into an action is equal to the number of executions of the command. This kind of execution is found in older interpreters for Bourne shell scripts for Unix<sup>TM</sup>, for example.
2. It may read the code, recode it into a more convenient form, and then use an interpreter for the more convenient form to execute the code. This approach is used for **Perl** scripts, for example.
3. It may read the code, and recode it into highly stereotyped machine code, where each high-level command of the recoded form is also a machine code sequence whose effect is to carry out the command (threaded code). This approach is used for **Forth** source programs, for example.
4. It may read the code and compile parts of it into optimized machine code on a cached basis using a 'just in time' compiler (a JIT). This approach is used by the **Smalltalk 80<sup>TM</sup>** and **Java<sup>TM</sup>** virtual machines, for example.

And, of course, any combination of the above approaches may also be used. For example, a Java<sup>TM</sup> virtual machine may both interpretively execute bytecodes and employ a JIT.

However it is accomplished, we call any program which uses such an approach, or a combination of such approaches, in which the executable code is not machine code and/or is considerably modified prior to execution, and is executed by the program which reads the code, an *interpreter*. As a corollary, we use the term *script* in a very broad sense: we consider Java *.class* files or *.jar* files to be scripts, for example.

*Two Sources, One Problem, One Solution:* Although *machine executables* and *scripts*, as defined above, are superficially different — the former are executable with very limited change in format; the latter only with significant changes — they represent a single problem: they are executable, and when programmed by a cracker and received from outside a given computer system, they may cause substantial damage.

The largest problem with alien executables is that code is very much the same kind of thing irrespective of its point of origin or the purpose of the person or persons who originally created it. Schemes based on cryptographic *code signing* can provide highly effective protection for ensuring that the received code is the code that a known sender intended to be received, but such cryptographic schemes require additional infrastructure, one must be able to trust the sender, and the requirements of such an approach limit the ways in which we can work.

Moreover, merely marking executables superficially with identifying tags is insufficient. For example, script files on Unix<sup>TM</sup> or GNU/Linux systems may begin with the 'shebang' notation, which indicates what program should be used to interpret the executable code. We could extend this with identifying information which could be checked as interpretation begins, but this would be quite inadequate: *parts* of executables may be hostile. For example, a virus attaches itself to an existing executable and makes its hostile action part of the behavior of the executable to which it is attached. In this case, most of the code is benign: only an added part — normally a small portion of the code — is hostile.

We cannot identify hostile code in general. That is well beyond the state of the art. Computer programs cannot understand a programmer's intent, and even human programmers cannot guarantee to do so. The best we can do, then, is to have a reliable way of identifying code from an external source, whether it is an entire executable or a fragment of an executable. That is, we must have a reliable way of identifying *alien code*.

- Our method for protecting against *viruses* is *almost identical* to our method of defense against buffer overflow attacks: it simply applies exactly the same kind of encoding method in a different, but analogous, context.
- Our method for protecting against *alien programs* is a *natural extension* of the above-mentioned virus-foiling defense.

In any case, the above method of protection against viruses and other hostile code fragments, and its corollary, the protection against hostile programs, can be applied in quite similar ways, whether we are protecting against potentially hostile alien *machine* code using a protecting *loader*, or against potentially hostile alien *script* code using a protecting *interpreter*.

**Protecting Against Alien Code Fragments:** Our protection against potentially hostile code fragments, whether the code is machine code or script code, is very much derived from our approach to buffer overflows.

In a buffer overflow, we ensure that data which overflows, if specially written by a cracker to perform some hostile or illicit action, will fail to do so since the overflow will be written with one coding unknown to the cracker, and read with another encoding (or for system data or code, with no encoding). That is, we ensure that the interpretation of what the cracker provided will be *incorrect* from the cracker's point of view, so that it will fail to perform the function intended by the cracker.

This immediately leads to an effective way to protect against alien code fragments appended, prepended, inserted, or overwritten into a legitimate piece of local executable code (whether it is a machine executable or a script). We simply use an *encoding* for local scripts or machine executables. For this purpose, we would recommend an encoding such as the 16-byte group encoding described in section 3.2.3 on p. 8, which provides extremely high diversity and is infeasibly difficult to attack remotely by brute-force. Such an encoding might have overhead a bit high for normal deployment in the **protector** modifications to an application, but in connection with a protecting *loader* or *interpreter*, which must obtain their inputs by messaging or from an I/O device, would have completely negligible overhead.

The entire local machine executable or script is so encoded, and the protecting loader or interpreter is derived from the normal one by modifying it to take its machine executable or script input in this encoded form. (Programs which produce machine executables and scripts may be correspondingly modified to write them in this encoded form where necessary.)

Of course, all of the executables to be executed by protecting loaders and interpreters must be correspondingly modified to have the proper encoding so that they will work.

As a result, *any added alien fragment cannot be correctly executed or interpreted, since it is incorrectly encoded for any executable code on the system on which it is added.* If it occurs where machine code is expected, it is not proper machine code; if it occurs where script code is expected it is not proper script code. Hence any actions written into any such fragment (say a virus) by a cracker *cannot occur*. Its potentially hostile actions are absolutely foiled. If such a fragment is created, in an *insider attack*, with knowledge of a particular system's encoding for such purposes, it will not work on any other system due to the diversity of encodings used among systems. Indeed, if it is created with knowledge of the encoding used at a particular time, it probably will not work at another time on the same system, if the encodings are refreshed at random and sufficiently frequent intervals.

(It would be straightforward to install this kind of modifying capability in the **protector** as an additional capability, since it need only identify the places in the code where the input or output occur, and apply the appropriate encoding on write or decoding on read. That is, it is a simple extension to the **protector** to enable it to create protecting loaders and interpreters and associated machine executable and script writing programs, as well as to carry out its normal function of protecting applications against buffer overflow attacks.)

**Protecting Against Alien Programs:** There are two defenses against potentially hostile alien programs by protecting loaders and protecting interpreters:

1. Refuse to execute them at all. In that case, the defense against alien code fragments described above already achieves it — if the defense is installed as described above, with no extensions, alien programs will certainly not execute since, from the protecting loader's or protecting interpreter's point of view, they contain incomprehensible garbage.

If programs are to be imported over the Internet, they should then be imported carefully, with adequate checks on protections such as code signing, and then the current encoding for such executables should be applied so that they will work properly on the importing system.

2. Execute them, but at a reduced level of privilege, or in a restricted environment (which is called a *sandbox*). In this case, the modifications made to loaders and interpreters must be slightly extended to provide an additional bit of functionality.

The extension, in case we want to permit alien programs to execute, but in a restricted fashion, is described here.

In this case, the protecting loader or protecting interpreter attempts to deal with the program using the system's current encoding. If this succeeds, it is dealing with a non-alien program and proceeds normally.

However, if it fails, the extension is to try reading the program *without* decoding it. If this succeeds, the program is an *alien program*. In that case, the protecting loader or protecting interpreter proceeds normally, *except* for the way in which the code is executed. It is either executed in a *sandbox* or with reduced privileges, so that its restricted environment or its restricted privileges prevent it from harming the system into which it has been imported.

Again, it is straightforward to add the ability to convert a loader or interpreter into a protecting loader or protecting interpreter with this kind of privilege- or environment-restricting form to the **protector**. What the **protector** must do is to locate the initial input of the executable code (whether machine code or script code) and insert a check: does this look like an encoded executable or an unencoded one? This is then used to set a flag, not accessed by the remainder of the loader or interpreter code except as noted below, which indicates whether the executable was encoded (non-alien) or unencoded (alien). The **protector** locates the point where the loader or interpreter would initiate execution of the executable code, and modifies the code so that, wherever this occurs, the first step is to ensure that the privilege level is reduced or the *sandbox* environment is established.

With this added capability, the **protector** can ensure that, whether alien code is in the form of an alien code fragment or an alien program, the system on which the code is executed is protected against hostile actions which the alien code might perform, whether by preventing any execution of such actions (for a fragment or a program whose execution is simply refused) or by reducing its capabilities using a *sandbox* or reduced privileges so that such hostile actions cannot be effective.

**We claim:**

1. A method of computer program protection against attack by modifying the program, so that the entire program or the portions of the program data associated with arrays which might be buffers, are encoded.
2. A method of computer program protection as in 1, in which different buffers, or arrays which might be buffers, within a program are given different encodings.
3. A method of computer program protection as in 1, in which different applications of the protection to a given program employ distinct encodings for each buffer, so that an attacker's knowledge of the encodings in a program used for one application of the protection does not provide a means of attacking the same program protected by distinct application of the protection.
4. A method of computer program protection as in 3, in which the method of application is re-applied periodically to a program used at a given site so that an attacker's knowledge of the encodings used at one time is invalidated for future attacks by the resulting changes in the encodings chosen when the protection is re-applied.
5. A method of computer program protection as in 2 or 3, in which the encodings chosen for programs, or for portions of arrays which might be buffers, are different from one user site or geographical location to another, so that an attacker's discovery of the encodings involves searching among alternatives, and so that an attacker's knowledge of the encodings in a program used at one site does not provide a means of attacking the same program at another site since the protection is applied to make the encodings different at the other site.
6. A method of computer program protection as in 1, in which portions of program data which might be buffers are encoded, but in which system data, such as the linkages of routines, library support data not used by an application, and other forms of system information, are not encoded.
7. A method of computer program protection as in claim 1, in which the encoding or encodings used to encode the program or the data which might be buffers is or are chosen to be non-identity bijective encodings.
8. A method of computer program protection as in claim 1, in which the encoding or encodings used to encode the program or the data which might be buffers is or are chosen to be dependent only on the data and the address of the data.
9. A method of computer program protection as in claim 6 or claim 2, in which encodings are so applied that, when an encoded array or buffer overflows, the overflowing data is encoded with the encoding of the array from which it overflows, thereby causing it to be incorrectly encoded for system data (which is not encoded) or neighboring encoded arrays, so that attacks *via* such overflows are foiled by the inappropriate encoding of the overflowed data.
10. A method of computer program protection as in claim 2 in which the encoding to be used for a particular explicit or implicit array in memory is obtained by additional data added to the pointer or pointers used to access such arrays.
11. A method of computer program protection as in claim 2 in which the encoding to be used for a particular explicit or implicit array in memory is obtained by maintaining a separate mapping data in which the beginnings and endings of arrays and their associated encodings are stored.
12. A method of computer program protection as in claim 11 in which the mapping data is maintained in the form of a balanced tree.
13. A method of computer program protection as in claim 11 in which the balanced tree is provided with static element space for static arrays, or stack element space for arrays on the run-time stack, or heap element space for

- arrays in the heap, so that the same kind of allocation of space occurs for the element space as for the arrays for which they provide encoding information.
14. A method of computer program protection as in claim 11 in which the memory mapping is extended with encoding caches containing one or more most recent mappings, so that locality of reference to encoded data will result in faster determination of its associated encoding.
  15. A method of computer program protection as in claim 6, in which the identification of buffers is performed by partitioning the program into application and system, and by identifying calls to system routines moving data in and out of application data arrays, thereby determining that they are buffers.
  16. A method of computer program protection as in claim 15, where regions of data are considered to be arrays either if they are accesses as arrays, or if they are accessed using pointer incrementing or decrementing in connection with a region of memory.
  17. A method of computer program protection as in 1, in which encodings are chosen to preserve the size of the encoded items, whether instructions or data.
  18. A method of computer program protection as in 1, in which encodings are chosen to be implementable by straight-line (i.e., non-branching, non-conditional) instruction sequences.
  19. A method of computer program protection as in 1, in which encodings are chosen so that fetch and store sizes of the program, i.e., the sizes of the blocks of storage fetched and stored, prior to the changes required for protection, are preserved in the protected program.
  20. A method of computer program protection as in claim 1, in which the encodings are computed using formulas involving one or more small constants and the address of the encoded information.
  21. A method of computer program protection as in claim 20, in which the small constants are byte values.
  22. A method of computer program protection as in claim 1, in which entire programs are encoded in their machine executable code file or script code file forms, so that fragments overwritten onto the program, or prepended to it, or appended to it, or inserted into it, are not encoded in the same fashion as the remainder of the program, which is then used in combination with a modified loader or interpreter which accepts programs in the modified form by decoding it on read, so that any attack by such a fragment is foiled because the loader or interpreter will read it as incorrectly formatted.
  23. A method of computer program protection as in claim 22, in which the system on which the modified loader or interpreter operates is protected by consistently encoding all, or many, of the programs to be loaded and executed, or interpreted and executed, so that they are recognized as belonging to that system..
  24. A method of computer program protection as in claim 23, in which different systems use different consistent encodings, so that a program imported from outside the system, whether encoded for another system or not, is generally not encoded with the consistent encoding used for the system into which it is imported.
  25. A method of computer program protection as in claim 22, in which the modified loader or interpreter is further modified so that, if it inputs a program

which is, in its entirety, not encoded, instead of treating it as incorrectly formatted, it reads it without the usual decoding, but executes the program in a restricted environment or with a reduced level of privilege, thereby protecting the system on which it runs from any hostile actions it might perform.

26. A computer program or computer programs for performing the method of any of claims 1 through 25.
27. A computer system comprising one or more processing units, storage media, and input and output devices, for performing the method of any of claims 1 through 25.

#### REFERENCES

- [1] David Aucsmith and Gary Graunke. Tamper-resistant methods and apparatus. United States Patent 5,892,899, 1999.
- [2] Bounds checking projects. Available at <http://gcc.gnu.org/projects/bp/main.html>, August 2000. A GNU-based project to add fine-grained dynamic bounds checking to the gcc compiler suite.
- [3] Stanley Chow, Harold Johnson, and Yuan Gu. Tamper resistant software – control flow encoding. Filed under the Patent Cooperation Treaty on August 18, 2000, under Serial No. PCT/CA00/00943, 2000.
- [4] Stanley Chow, Harold Johnson, and Yuan Gu. Tamper resistant software encoding. Filed under the Patent Cooperation Treaty on June 8, 2000, under Serial No. PCT/CA00/00678, 2000.
- [5] Stanley Chow, Harold Johnson, and Yuan Gu. Tamper resistant software – mass data encoding. Filed under the Patent Cooperation Treaty on April 12, 2001, under Serial No. PCT/CA01/00493, 2001.
- [6] Stanley Chow, Alec Main, Harold Johnson, Marcel Achim, and Yuan Gu. Computer system protection by communication diversity. Preliminary patent disclosure recently submitted to the Canadian Patent Office, 2001.
- [7] Frederick B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6), October 1993.
- [8] Christian Sven Collberg, Clark David Thomborson, and Wai Kok Low. Obfuscating techniques for enhancing software security. Filed under the Patent Cooperation Treaty in 1998 under Serial No. PCT/US98/12017; International Publication Number wo 99/01815, 1997.
- [9] Christian Sven Collberg, Clark David Thomborson, and Wai Kok Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, 1997.
- [10] Christian Sven Collberg, Clark David Thomborson, and Wai Kok Low. Breaking abstraction and unstructuring data structures. In *IEEE International Conference on Computer Languages*, pages 28–38, 1998.
- [11] Christian Sven Collberg, Clark David Thomborson, and Wai Kok Low. Manufacturing cheap, resilient and stealthy opaque constructs. In *Symposium on Principles of Programming Languages*, pages 184–196, 1998.
- [12] Debian home page. Available at <http://www.debian.org>, January 2002. Provides a list of top ten vulnerabilities in Debian releases.
- [13] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, California, 1997. IEEE Computer Society Press.
- [14] ICAT metabase. Available at the NIST web site at <http://icat.nist.gov/icat.cfm>, January 2002. Provides a list of vulnerabilities rated by their frequency in ICAT queries.
- [15] Janus. See <http://www.berkeley.edu/~daw/janus>. Janus is an open-source sand-box protection system from the University of California at Berkeley.
- [16] Donald E. Knuth. Balanced trees. In *The Art of Computer Programming*, pages 458–471. Addison-Wesley, Reading, Massachusetts, 1998. Description and analysis of AVL trees and their use in data mapping.
- [17] M. Mambo, T. Murayama, and E. Okamoto. A tentative approach to constructing tamper-resistant software. In *Workshop on New Security Paradigms*, pages 23–33, 1998.

- [18] Norton AntiVirus<sup>TM</sup>. See <http://enterprisesecurity.symantec.com/products/products.cfm>.
- [19] rdist vulnerability (Unix<sup>TM</sup>). See [http://www.dataguard.no/bugtraq/1996\\_3/0022.html](http://www.dataguard.no/bugtraq/1996_3/0022.html), July 1996. [8lgm]-Advisory-26.UNIX.rdist.20-3-1996.
- [20] The twenty most critical internet security vulnerabilities (updated): The experts' consensus. Available at the SANS Resources web site at <http://www.sans.org/top20.htm>, November 2001. Provides a list of the security weaknesses which are involved in the largest number of security breaches.
- [21] Security-Enhanced Linux. See <http://www.nsa.gov/selinux>. Security-Enhanced Linux (SELinux) is a research prototype from NSA of a Linux which applies access control rules to enhance system security, released for experimental purposes, primarily under the GNU Public License.
- [22] SourceGuard<sup>TM</sup>. See <http://www.4thpass.com>. SourceGuard is a commercial version of HashJava.
- [23] STAT Neutralizer<sup>TM</sup>. See <http://www.statonline.com/products/neutralizer/index.asp>. STAT Neutralizer is a site protection system combining rule-based access control, intrusion-detection using statistical profiles, and recognition of malware ancestry by pattern-matching on their code.
- [24] The tower of Babel. See **The Bible**, Genesis 11:1-9. When human beings rebel against God by attempting to construct the tower of Babel to reach heaven, God throws their efforts into confusion by diversifying their languages so that they can no longer communicate.
- [25] TRS resources. See the *resources* link available at (<http://www.cloakware.com>). Provides overviews of various aspects of TRS and diversity-based defense.
- [26] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report 12, Department of Computer Science, University of Virginia, 2000.