Improving Operation Time Bounded Mission Critical Systems' Attack-Survivability through Controlled Source-Code Transformation

Alban Vignaux, Arnaud Auguste,
Bogdan Korel, and Shangping Ren*
Dept of CS, Illinois Institute of Technology
Chicago, IL 60616
{avignaux, aauguste, korel, ren}@iit.edu

Kevin Kwiat
Cyber Science Branch
Air Force Research Laboratory
Rome, NY 13441
kwiatk@rl.af.mil

ABSTRACT

Mission critical systems often operate for limit time durations. For these systems, we subscribe to the notion that provisioning of security can be based on the expected duration of a system's mission. In this paper, we present a simple and safe $K-$variant approach to improve time-based mission critical systems' attack-survivability and provide formal analysis about $K-$variant system's attack survivability under M memory-based attack attempts. Our theoretical analysis supported by extensive simulations and a case study provide good evidences that the proposed approach may be in improving system's attack-survivability.

Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering

General Terms

Security, Reliability

Keywords

Mission Critical Systems, Time-Bound, Attack Survivability, N-Version Programming, K-Variant, Source Code Transformation

1. INTRODUCTION

N-Version programming emerged in the 1970's as a way to tolerate design and implementation faults in software. In particular, when a software fault is encountered during a program execution, the unaffected version will be able to provide computer systems with the ability to regain its initial operating capability.

As we know, technology attacks [8] often exploit programming errors or vulnerabilities that are accidentally or intentionally (life cycle attacks) introduced by software builders. N-Version programming is a useful defense strategy against such unanticipated or unimaginable (zero-day) technology attacks where concurrent, diverse, individually correct, and functionally equivalent programs are generated ahead of time to rapidly replace programs that have been successfully attacked. It provides coverage for design flaws, algorithm weaknesses, misconfigurations, scripting attacks, and data attacks. N-Version programming-enabled diversity invalidates the attacker's assumptions about targeted systems and thus blocks or disrupts cyber technology attacks. The most widely adopted state-of-the-art in synthetic diversity techniques to combat cyber attacks are: instruction set randomization [15, 19], address space randomization [23, 6], stack space randomization [6, 7], DLL base randomization [5], heap randomization [6, 7], encrypted instructions [17], calling sequence diversity [28], and system call renaming, to name a few. While being focused mainly on memory attacks, these techniques have nevertheless achieved success and are well accepted in the cyber defense community.

Software variations can be introduced at different phases of software development life cycle. However depending on the phase chosen, the final cost of the solution is different. The earlier in the life cycle the variations are introduced, the larger the variation and the cost. Hence, from reliability perspective, N-version at higher level of the software development stack is preferred [14]. However, from software engineering perspective, high level variations are much more expensive — any modification at one version could trickle a chain of reactions, including source code change. Even at the source code level, uncontrolled N-version is also difficult to verify and maintain their functional equivalences. We believe that such unfavorable reality is not due to N-version programming concept, rather, it is due to how the variants are introduced. Source code transformations have to be simple and the scope of modifications should be limited to avoid the introduction of new flaws in the program.

For mission critical systems, such as battle field control systems, the operation time is often rather short comparing to general purpose systems, such as web servers. However, during the short period, they need to be highly reliable and available even in the presence of austere malicious attacks. We subscribe to the notion that provisioning of security can be based on the expected duration of a system's mission. This is in contrast to the "fortress mentality" with its insistence upon impenetrable protection. Over a decade ago, Winn Schwarau [22] argued against such unattainable perfection when he conceptualized time-based security. To "stand the test of time" in Winn's sense means not to build, at the onset, a perfectly secure system that repels all attacks; instead, sufficient security resources are devoted to nullify attacks upon the system's weaknesses before those attacks can completely manifest. We therefore take the stance that if system weaknesses may be unavoidable, then system resources can be added so that damage due to those weaknesses can be avoided. Avoidance occurs as long as necessary to ensure the system's mission. We base our avoidance strategy on the mission's time-bound and then proceed with the necessary resource provisioning.

In this paper, we present a simple and safe approach that blends

*This research is supported in part by NSF under grants CNS 0746643, CNS 1018731, and CNS 1035894.
Approved for Public Release; Distribution Unlimited: 88ABW-2011-4498 dated 18 August 2011

the attractive variation features of N-Version programming with memory-based diversity to improve time-bounded mission critical systems attack survivability. In particular,

- we introduce memory-based diversity ($K-$variants) through controlled source code transformation;
- we provide theoretic analysis on the attack-survivability of a system with $K-$variants and under M attack attempts;
- we conduct extensive simulations to validate the effectiveness of the proposed approach and the theoretic analysis;
- we also provide a case study which not only supports our theoretical findings, but also may be viewed as a proof of concept for the K-variant paradigm presented in this paper.

The rest of the paper is organized as following: Section 2 uses an example to motivate the research. We formally define the problem the paper is to address in Section 3 and provide theoretic analysis in Section 4. The memory-based simple and safe source code transformation and experimental evaluations are presented in Section 5 and Section 6, respectively. We discuss related work in Section 7 and finally conclude the paper in Section 8.

2. MOTIVATING EXAMPLE

To motivate our research, consider a small program given in Figure 1:

```
void F(int index, int value) {
  int buffer[10];
  buffer[index] = value;
  ... };

main() {
  int x, i;
  ...
  read(i, x);
  if ((i > 0) && (i =< 10)) F(i,x);
  ... };
```

Figure 1: Original Program

This program has a buffer-overflow vulnerability. i.e., when the attacker enters the value of input variable `i = 10`, then because of incorrect guard at the `if`-statement the buffer-overflow occurs in function `F()`. This buffer-overflow will most likely cause a crash of the program, or be exploited by the attacker. However, if we introduce some unused memory in the neighborhood of the buffer, then the buffer-overflow occurs only in the redundant/unused memory that has no impact on the behavior of the program. Figure 2 shows a modified program of Figure 1 in which we introduce a dummy buffer (`int dummy_buffer[5]`) in function `F()`. This dummy buffer prevents the successful attack of crashing the system for input value of `i = 10`. Notice that the introduction of this extra redundant memory in function `F()` does not change the functionality of the program.

```
void F(int index, int value) {
  int dummy_buffer[5];
  int buffer[10];
  buffer[index] = value;
  ... };
main() {
  int x, i;
  read(i, x);
  if ((i > 0) && (i =< 10)) F(i, x); };
```

Figure 2: Transformed Program by Adding a Dummy Buffer

In this example, we have applied a simple behavior preserving source-code transformation to the program of Figure 1, by inserting a dummy buffer. The resulting transformed program of Figure 2 is semantically equivalent to the original program. In this paper, the transformed program is referred to as a variant of the original program. The goal of this research is to investigate ways of behavior preserving source-code based transformations which can be applied to the original program to minimize the chances of successful attacks on the system consisting of the original program and one or more of its variants (K-variants) within a given time-frame. Ideally, a successful attack on the original program should not be successful when it is applied on the transformed variant(s). As a result, the attacker needs to spend more time and resources during the attack not only on the original program but also on its K variants. This additional time for the attacker may increase the chances of a successful completion of the mission by at least one of its variants.

By using different behavior preserving transformations in different places in the source code, we can automatically generate a large number of variants of the original program. Before a mission critical system is launched, K variants are randomly selected for the mission, i.e., a different set of variants is used for each mission. This may reduce the chances of successful attacks for subsequent missions.

However, the behavior preserving source-code based transformations must be simple and safe, where by a safe transformation we mean a transformation that does not introduce any side-effects to the original program. By a simple transformation we mean a transformation that does not require any additional transformations (changes) in order to ensure the semantic equivalence. For example, the transformation of Figure 2 is a simple transformation because adding a dummy buffer does not require any additional changes to the source code. On the other hand, a transformation that, for example, changes a name of a variable may require many changes in the source code where the variable is referenced. Such a complex transformation may not be safe because it requires many additional changes in the source code. As a result, a significant effort for retesting of such variants may be required.

In summary, the idea of our approach is to automatically restructure an existing body of source code of an original program without changing its external behavior by applying a series of small behavior preserving transformations that may result in a different internal program behavior during the attacks.

3. PROBLEM FORMULATION

In this section, we formally define the problem the paper is to address. It is worth pointing out that the intend of this paper is *not* to investigate how to prevent attacks, rather to develop a simple, less expensive, but effective software approach to enhance mission critical systems' operation time in the presence of attacks. We focus on memory exploitation attack in this paper.

System Model

We assume that to increase a mission critical system' availability and reliability in a hostile environment, K different variants $(V_0, V_1, \cdots, V_{K-1})$ of the system' software modules operate concurrently. These K variants are transformations from an initial version V_0 by using memory-based *simple semantic preserving source-code transformations*.

Attack Model

We assume the best case scenario for the attacker. In particular,

- The attacker is aware of the existences of all $K-$variants.
- The attacker makes all attack attempts concurrently on all $K-$ variants.
- Each attack attempt takes T_{attack} time.

- For the time duration (T_{sys}) in which the system operates, an attacker can make at most $M = T_{sys}/T_{attack}$ attack attempts.
- By a successful attack we mean that the attacker is able to crash all variants within the M attempts; if at least one variant survives all M attacks, it is considered as an unsuccessful attack.
- Attacks are random and the attack address is uniformly distributed within a given memory space.

Notations

To simplify the discussion, we first introduce the notations that are used in the paper.

- N = the size of the memory potentially under attack. We assume that the size of the memory under attack is the same for all $K-$variants.
- n = the size of the vulnerable memory and it is located within the memory under attack and $n \leq N$. We assume that a memory unit at some address is vulnerable if its modification by the attacker causes a failure of the variant under attack.
- r = the percentage of vulnerable memory with respect to the total memory under attack for a given variant, i.e., $r = \frac{n}{N}100\%$
- M = maximum number of attack attempts an attacker can make.
- K = number of software variants a system has.
- $P_u(K, M)$ = the probability that an attacker fails to crash all the $K-$ variants with M attempts. Such an attack is considered as an unsuccessful attack. We consider $P_u(K, M)$ as a measure of the survivability of the system under attack.
- $P_s(K, M)$ = the probability that an attacker successfully crashes all the $K-$ variants after maximal number of M attempts, i.e., the probability of the successful attack.
- $O(addr, i, j)$ = a Boolean value that indicates whether vulnerable memories of variant i and j overlap at address $addr$. $O(addr, i, j) = 1$ when there is an overlap; otherwise, its value is zero.

Problem Formulation

Under the system and the attack models presented above, we formally define the problem we are to address:

PROBLEM 1. *Given a mission critical system with executable software memory size N and vulnerable memory size n, assume $K-$variants are created through simple semantic preserving source code transformations and operate concurrently with their initial version, can the new system survive M attacks from malicious attackers? In other words, what is $P_u(K, M)$ under the given system and attack models?*

□

4. SYSTEM SURVIVABILITY ANALYSIS

In this section, we formally analyze the influence of different factors on the attack survivability for the $K-$variant approach presented in this paper for mission critical systems.

Consider two program variants V_0 and V_1, where V_1 is obtained through a program transformation discussed in Section 2, i.e., adding a dummy buffer. The two variants' memory spaces are shown in Figure 3.

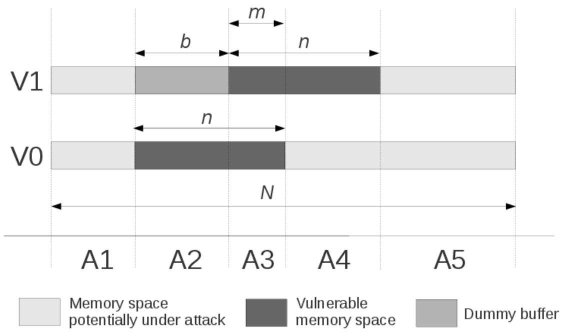

Figure 3: Memory Space for Two Variants

It is not difficult to see that, with only one version, i.e., V_0, the probability of an unsuccessful attack after M attempts (or the system's survivability of M attack attempts) is given below:

$$P_u(1, M) = (1 - \frac{n}{N})^M \tag{1}$$

When a new variant (V_1) is introduced, depending on the locations of the vulnerable memory of V_0 and V_1, the probability of unsuccessful attacks can be different when the relationship between of the two variants' memories changes. We discuss each case in detail:

Case 1: the vulnerable memory space of V_1 is the same as in V_0. In this case, the probability of an unsuccessful attack after M attempts is given below (2):

$$P_u(2, M) = P_u(1, M) = (1 - \frac{n}{N})^M \tag{2}$$

Case 2: The vulnerable memory space of V_1 is different from V_0, but there is an overlap of size m. However, the vulnerable memory space in V_1 is within the memory under attack as shown in Figure 3.

$$P_u(2, M) = 2(1 - \frac{n}{N})^M - (1 - \frac{2n - m}{N})^M \tag{3}$$

Case 3: The vulnerable memory space of V_1 does not overlap with the vulnerable memory space of V_0, i.e., $m = 0$, but it is within the memory under attack. We have

$$P_u(2, M) = 2(1 - \frac{n}{N})^M - (1 - \frac{2n}{N})^M \tag{4}$$

Case 4: The vulnerable memory space of V_1 is outside the memory under attack. Clearly, the probability of unsuccessful attack is 1, i.e.,

$$P_u(2, M) = 1 \tag{5}$$

Case 5: The vulnerable memory space of V_1 does not overlap with the vulnerable memory space of V_0, and is only partially with size q within the memory under attack. In this case, the unsuccessful probability is:

$$P_u(2, M) = (1 - \frac{n}{N})^M + (1 - \frac{q}{N})^M - (1 - \frac{n + q}{N})^M \tag{6}$$

Case 6: The vulnerable memory space of V_1 is different from but overlap with the vulnerable memory space of V_0 and it is only partially with size q inside the memory space under attack as shown in Figure 4. It is not difficult to see that this is the most general case from which **Case 1** to **Case 5** can be derived. The most general form of unsuccessful attack probability, i.e., attack survivability, is given by (7):

$$P_u(2,M) = (1-\frac{n}{N})^M + (1-\frac{m+q}{N})^M - (1-\frac{n+q}{N})^M \qquad (7)$$

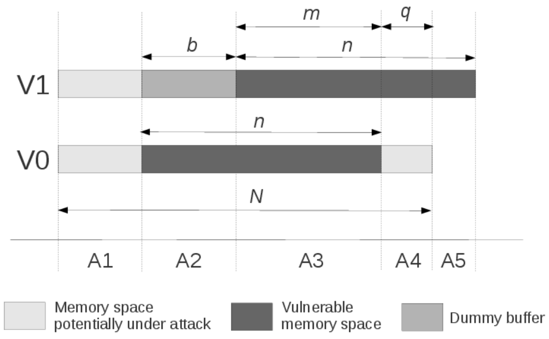

Figure 4: Memory Space for Two Variants when Partial Vulnerable Space Is Outside the Attack Range

Figure 5 demonstrates the influence of the dummy buffer size on the attack survivability based on the theoretical model presented by formula (7) when there are two variants and the maximum number of attack attempts (M) allowed is 50. From the figure, it is clear that for a given number of attack attempts, if the dummy buffer size is large enough, the system's survivability approaches to 1. Furthermore, as the vulnerable memory ratio, i.e., r, increases, in order to have high survivability, we have to introduce larger size of dummy buffer.

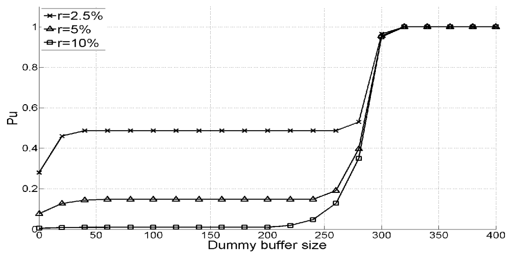

Figure 5: Attack Survivability vs Dummy Buffer Size

As can be seen from Figure 5 all three curves have some similarities, i.e., they have two flat parts. In fact, the first flat parts of the curves are for the case where there is no memory overlapping between variants with respect to the vulnerable memory. The second flat part in these curves for which $P_u = 1$ represents the situation where the vulnerable memory in the second variant is shifted outside of the attack range. These results clearly demonstrate that from the theoretical point of view it should be sufficient to introduce only one additional variant provided that the dummy buffer shifts the whole vulnerable memory outside of the "attacker's reach". This may work for some types of attacks, but for many other types, the memory under attack is the whole memory in the system. Therefore, it is not possible to shift out completely the vulnerable memory. However, it is possible to improve the probability of an unsuccessful attack (i.e., improve the attack survivability) by introducing dummy buffers at different places of a program so that the vulnerable memories in different variants do not overlap totally.

With $K-$variants, the best scenario for defender is that the vulnerable memory of different variants do not overlap with each other. Under this case, the system's attack survivability $P_u(K,M) = 1 - P_s(K,M)$, where $P_s(K,M)$ is given by (8):

$$P_s(K,M) = K \times \sum_{i=1}^{M-K+1} (1-\frac{K\times n}{N})^{i-1}(\frac{n}{N}) \times P_s(K-1, M-i) \qquad (8)$$

where $P_s(0,M) = 1$, $P_s(K,0) = 0$, and we assume $N \geq K \times n$.

Based on equation (8), Figure 6 shows the relationships between the number of variants and P_u with respect to different vulnerable memory ratio r assuming there is no memory overlap between variants. From the figure, it is not difficult to see that the program's inherent vulnerability r has a significant influence on the number of variants needed to achieve a specific attack-survivability.

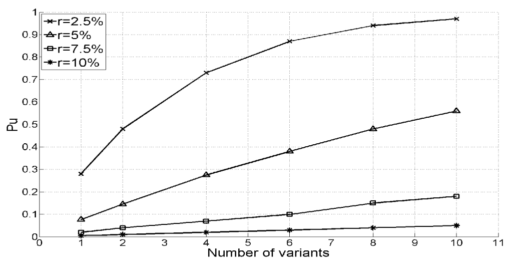

Figure 6: Survivability vs Number of Variants with Respect to Vulnerable Memory Ratio r

On the other hand, based on equation (8), Figure 7 shows the relationships between the number of variants and P_u with respect to different maximum numbers of attack attempts assuming the vulnerable memory ratio $r = 2.5\%$. It is not difficult to see that the maximal duration of the attack, represented by M, has a significant influence on the number of variants needed to achieve a specific attack-survivability.

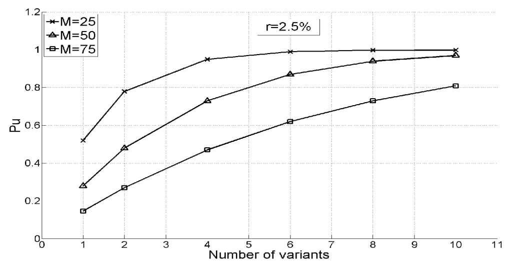

Figure 7: Survivability vs Number of Variants with Respect to Number of Attack Attempts M

These results suggest that the vulnerable memory space for different variants should not overlap, or at least the overlap should be minimized between variants. With our approach, as random dummy buffers of different size are introduced in different places of the source code of the system, the attacker may not know exactly where the vulnerable memory spaces for the next variants are exactly located. As a result, the survivability of the $K-$variant system may significantly increase.

5. CREATING K-VARIANTS THROUGH SIMPLE AND SAFE SOURCE CODE TRANSFORMATION

As we discussed in the previous section, the challenge is to identify simple and safe source code transformations that may (1) minimize the vulnerable memory overlaps between variants and/or (2) reduce the vulnerable memory size in variants. Transformations with such properties should enhance the survivability. In this section, we present three simple source code transformations that hopefully may reduce the memory overlap: (1) adding a dummy buffer(s), (2) dummy heap request(s), and (3) changing the order of heap-memory requests. Not every source code transformation, e.g., transformations used in refactoring, is appropriate for enhancing survivability of the $K-$variant system. For example, a transformation that changes a name of a function has no effect on survivability. The research challenge is to identify a set of transformations that may significantly reduce the chances of a successful attack within a given time-frame. The major advantage of source code transformations is that it gives an user some degree of control over generation of types of variants, as opposed to other methods, e.g., randomization, where the user has not much control over generated variants.

5.1 T1 - Adding dummy buffers and variables

This transformation consists in adding extra buffer (or a variable) declarations within the source code, like in the following example:

```
/* Original Variant */
int vulnerable_function(char * s){
  char dest[20];
  strcpy(dest, s);}

/* Transformed Variant */
int vulnerable_function(char * s){
char dummy_variable[100];
  char dest[20];
  strcpy(dest, s);}
```

This transformation helps to protect against buffer overflow as it adds more memory, depending on the size of the dummy buffer, between the vulnerable buffer and the return address of the function. Thus the input necessary to exploit the vulnerability is not the same. This type of transformations can easily be supported by a tool. Such a tool needs to identify potential places in the source code that may be vulnerable for an attack, i.e., guarding the return address, identifying vulnerable buffers (e.g., buffers used by unsafe functions). The dummy buffers can be added to "protect" local or global buffers/variables.

Issues that need to be determined when using such transformations are:

- How many dummy buffers shall be inserted into a variant(s),
- Where the dummy buffers should be inserted, and
- What are the sizes of dummy buffers

For each variant, different number of dummy buffers (of different sizes) can be inserted in different places of the source code. This generation of variants can be done randomly or semi-randomly, where the user can influence some of these factors, e.g., indicating potential places where dummy buffer insertions should be made.

5.2 T2 - Dummy heap requests

The purpose of this type of transformation is to insert redundant heap memory between heap memory spaces that are used by the system. For example, in the code below in the transformed variant a redundant heap memory request is inserted $dummy\_buf = malloc(sizeof(char) * 100);$. As a result, a dummy buffer is inserted between the heap memory space pointed by $sensitive\_data$ pointer and $dest$ pointer.

```
/* Original Variant */
int vulnerable_function(char * s){
  char * sensitive_data;
  char * dest;
  sensitive_data = malloc(sizeof(char)*100);
  dest = malloc(sizeof(char)*20);
  strcpy(dest, s);}

/* Transformed Variant */
int vulnerable_function(char * s){
  char * sensitive_data;
  char * dest;
  char * dummy_buf;

  sensitive_data = malloc(sizeof(char)*100);
  dummy_buf = malloc(sizeof(char)*100);
  dest = malloc(sizeof(char)*20);
  strcpy(dest, s);
  free dummy_buf;}
```

This is a relatively simple transformation and it can be easily supported by a tool. Potentially such dummy heap memory requests can be done before or after each "actual" heap memory request. Similar issues that need to be determined when using such transformations are:

- How many dummy heap memory need to be inserted into a variant(s),
- Where the dummy heap memory should be inserted, and
- What are the size of dummy heap memory requests

One of the issues that may need to be considered when using such a transformation are memory leaks. One needs to determine if unused heap memory needs to be returned to the heap. If a sufficient heap memory is available to run the system, then the memory leak is an acceptable side effect of increased survivability. However, if memory leaks are not acceptable, we have to release the memory allocated to prevent memory leaks. The transformation has to dispose the dummy allocated memory in every independent path of the function. This may be more complex than transformation T1, however, this transformation is still quite simple.

Another issue is that the frequencies of dummy memory requests may need to be controlled, e.g., when the heap memory requests are in a loop. In such a case, the number of dummy heap requests may be controlled by a conditional statement, e.g., the following statement can be inserted:

```
  if random_request()
      dummy_buf = malloc(sizeof(char)*100);
```

where each time when the if-statement is executed, the random function $random\_request()$ decides if the dummy heap memory request should be made or not.

5.3 T3 - Changing the order of heap-memory requests

The purpose of this type of transformations is to change the order of heap requests as shown in the code below.

```
/* Original Variant */
int vulnerable_function(char * s){
  char * sensitive_data;
  char * dest;
  sensitive_data = malloc(sizeof(char)*100);
  dest = malloc(sizeof(char)*20);
  strcpy(dest, s);}

/* Transformed Variant */
int vulnerable_function(char * s){
  char * sensitive_data;
  char * dest;
  dest = malloc(sizeof(char)*20);
  sensitive_data = malloc(sizeof(char)*100);
  strcpy(dest, s);}
```

This last transformation seems easy but needs extra precaution because new sort of problems can arise from this modification. As an example, between the old and new location of the memory allocation, some code can directly reference this variable. If this is not a case, then this transformation can be safely used. However, for a general case this type of transformation still requires more investigation.

6. CASE STUDY

The goal of this case study is to investigate the effectiveness of source code transformations (extra memory injections) discussed in this paper on the potential reduction of successful attacks for all $K-$variants within a given time-frame. In particular, we are to (1) evaluate the relationship between the number of surviving variants and the size of dummy buffers that are transformed into the initial variants, and (2) evaluate the number of attempts needed to have a successful attack. By a successful attack we mean that the attacker is able to compromise all $K-$variants. If there is one variant that is not compromised within a given time-frame, the attack is considered un-successful.

Case Study Settings

In this case study we have concentrated on the buffer overflow vulnerabilities. We have selected several programs for this study. For each program we have inserted some buffer-overflow vulnerabilities. Notice that it is assumed that these vulnerabilities are not detected during the regular testing/QA process. For each program we have created three variants by using source code transformations (i.e., injecting extra memory) discussed in this paper. In addition, we assume that an attacker can concurrently attack all the variants. In order to obtain statically significant results, each random attack is repeated 1,000 times. The time-limit is measured by the number of allowed random attacks (probes).

Case Study Results

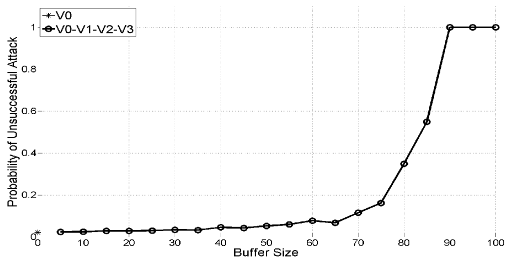

Figure 8: Probability of unsuccessful attacks vs dummy buffer size

Figure 8 plots the probability change [1] of unsuccessful attacks when the dummy buffer size increases for one of the programs. As we can see from the figure that the probability of unsuccessful attacks, i.e., the system's survivability, slightly increases as the dummy buffer size increases until the dummy buffer size reaches to 70 when the system's survivability sharply reaches 54% for a buffer size of 85. This is due to the program variant 2 and 3 becoming more resilient to random attack. The vulnerable space shrinks as the dummy buffer size increases. Finally with a dummy buffer size of 80 the two variants are no longer vulnerable. This trend was observed in all programs under investigation.

[1]The mean value is used in calculating the probability of unsuccessful attacks for each buffer size.

Figure 9 depicts the relations between the number of attacks needed to compromise individual variants compared to the size of the dummy buffer for different variants for one of the program under study.

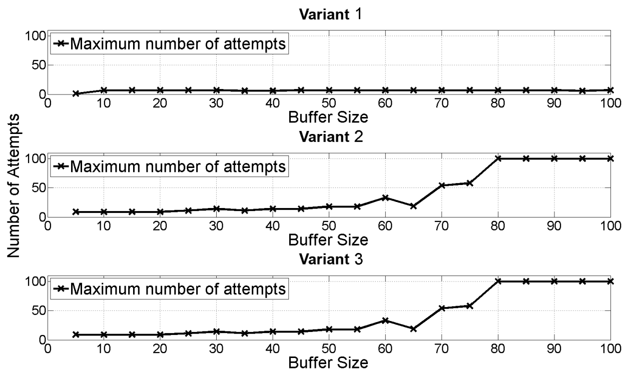

Figure 9: Number of attempts compared to dummy buffer size

We can observe that for this program variant $V1$ is not very useful as the maximum number of attempts needed to break the system is low and steady all along the experience. However for the last two variants, as the dummy buffer size increases the vulnerable area decreases and so the maximum number of attempts needed to compromise the system reaches 100, i.e., the maximum allowed.

The results of the small case study confirm that the memory based small transformations can significantly enhance the survivability of the K-variant system.

7. RELATED WORKS

In this section, we discuss related work from three different aspects, namely, security of mission critical systems, N-version programming, and specific buffer overflow protections.

Security of critical system

For mission critical systems, such as battle field control systems, the operation time is often rather short comparing to general purpose systems, such as web servers. They are also limited by physical constraints, such as space and size constraints. The short operation time and physical constraints often inhibit the system's ability to repair themselves during mission time. But on the other hand, similar to general purpose systems, critical systems are also prone to many types of errors. Furthermore, due to its criticality and short operation time, a mission critical system often faces higher probability of transient failures [18].

To deal with problems of reliabilities, modern critical systems incorporate fault-tolerant techniques to get a more robust system protected against faults. Fault tolerance does not aim at removal of all vulnerabilities in a program, rather at ensuring that these vulnerabilities do not affect the correct execution of the program. Thus, a software unit is fault-tolerant if it can continue delivering the required service, i.e., supply the expected outputs with the expected timeliness, despite the presence of fault-caused errors within the system itself. Getting a high degree of tolerance in a program in not easy. Errors need to be detected with the most possible brevity, and the propagation of erroneous information through the system have to be avoided [20].

Research studies have pointed out that most system vulnerabilities detected are due to bad programming habits [2]. For instance, fail to preserve SQL query structure or constrain operations within the bounds of a memory buffer, improper access control, hard-coded password, etc, belong to the category. Attackers can often easily take advantages of these weaknesses and cause system to crash or even take the control of the system. Buffer overflow, in particular, is an program anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. It is currently one of the most common vulnerabilities and be maliciously exploited by attackers.

To date, many attack techniques have been developed to exploit memory errors: stack smashing attacks [1], return-to-libc attacks [11], format-string attacks [25], data modification attacks, heap overflow attacks [16], integer overflow attacks [9], to name a few. Many counter-attack techniques have emerged from system security community to prevent systems from becoming victims of such exploits [24, 12]. However, because these preventive measures are generally applied, they neglect the specific constraints placed upon the attackers and defenders.

Our work aims at improving the security of a specific type of mission critical systems of which the operation time is bounded. The implication of such time bound is that attackers have a limited time to exploit a weakness in the program.

N-version programming

Currently, one of the main technique for achieving fault tolerance is N-version programming, a technique based on diversity which aims at employing several alternate versions of a program, all responding to the same given specification. Basically, it consists of executing multiple versions of the same application in parallel; each receives identical inputs and each produces its version of the required outputs. The outputs are collected and submitted to a decision algorithm that selects the output to be used by the system. Some methodologies have been proposed to effectively apply the N-version programming technique to software systems [4, 21].

Unfortunately, the N-version programming methods often lead to a growth of costs (men power, time and space, and other resources) and a higher complexity in terms of design and programmability [3]. The different versions must have maximal independence of design and implementation. It implies the use of diverse algorithms, programming languages, compilers, design tools, and implementation techniques, etc. It can also imply the employment of independent (noninteracting) programmers or designers, with diversity in their training and experience. Moreover, implementation of N-version fault-tolerant software requires special support mechanisms that need to be specified, implemented, and protected against failures. For example, a decision algorithm required for the approach is itself difficult to implement [10]. Finally, maintenance of a N-version program remains a challenging issue: a single modification could affects all versions.

Different from N-version programming, the software $K-$variants we propose can be obtained from the initial version of a program by simple and safe source-code transformations. Comparing to N-version programming, our source code transformation is easier to implement and has less costs in terms of programmability, efforts, and execution time.

Buffer Overflow Protection

StackGuard and ProPolice

StackGuard [13] is an extension added to the GCC compiler to detect and thwart stack smashing buffer overflows. The effect is transparent and the flow of a program is kept. However, the stack frame is modified with the add of so called 'canary' value between the control data and the buffer. It is initialized after the control values (saved registers, frame pointer, return address) are saved and checked right before the control values are restored [27]. Later, ProPolice appears as an enhancement of the StackGuard protection. It differs by protecting more than the return address: all registers saved in the function prologue are also protected.

StackGuard is limited to one type of attack: stack smashing. It may not be effective in preventing other types of attacks. The idea of StackGuard is similar to transformation T1 which introduces dummy buffers in the source code whereas StackGuard introduces extra memory during compilation. However, transformation T1 is more general because it is not limited only to stack memory. In addition, transformations T2 and T3 provide other ways of protection related to the heap memory that are not supported by StackGuard. Finally, StackGuard users have a very limited control during the process of generation of different variants of the system as opposed to our approach that provides a higher level of diversity between variants.

Address space randomization

Another way to prevent systems from memory errors is memory address space randomization. This technique randomly arranges the positions of key data areas of a process's address space [23], such as the base of the executable, the position of libraries, the heap, and the stack. It aims at preventing the attackers from using the same attack code to exploit the same flaw in multiple randomized instances of a single software program by making it more difficult for the attacker to understand the structure of the program and to predict target addresses. In practical terms, randomization ensures that an attack that succeeds once may not succeeds a second time on the same program (a failed attempt typically making the program crash). This technique is particularly effective against large-scale attacks [5]. To date, different mechanisms, such as PaX ASLR (Address Space Layout Randomization) [26], implement this technique.

An approach to achieve address space randomization is address obfuscation [5]. It is a program transformation technique in which absolute locations of all code and data, and the relative distances between different data items are randomized. This can be realized through a combination of different transformations:

- Randomize the base address of memory regions (base address of the stack/the heap, starting address of dynamically-linked libraries, locations of routines and static data in the executable)
- Permute the order of variables/routines (order of local variables in a stack frame, order of static variables, order of routines in shared libraries or in the executable)
- Introduce random gaps between objects (random padding into stack frames, random padding between variables in the static area, gaps between routines and jump instruction to skip over the gaps)

As a result, the program's code is modified so that each time it is executed, the virtual addresses of the code and data of the program are randomized. This technique allows to protect a program against a wide range of attacks that exploit memory errors, and can easily be applied without modifying its source code. Even if it is quite effective, it can introduce runtime overheads and remains vulnerable to some specifics attacks.

Address obfuscation aims at randomizing code and data location at binary level. Different from address obfuscation, our approach allows source code level modification which gives application developer more control on the transformation processes.

8. CONCLUSION

Mission critical systems have more constraints than a general purpose system. Their execution is time limited and they have to remain available and reliable during this short operation period. As all software systems, they may contain some weaknesses and may be prone to attacks. Buffer overflow is one of the most common vulnerability that can be found and can be maliciously exploited by hackers.

In this paper, we developed a method that suits for critical systems' requirements (i.e. high reliability during a short operation time) and is easy to implement. By introducing dummy buffer in the source code of our program, we are able to modify the memory space structure so that an exploit which works on a version of a program does not work on another version. This technique has many advantages: it makes the modified system more robust against attacks: a hacker needs to spend more time to exploit several variants of a program rather than a single one, which is the most important factor in a critical system (where run time is limited). It is simple in terms of programmability and does not change the semantic of the program. Since modifications are made at code source level, it provides more control on the transformation process than a randomization of address space at a binary level.

However, this paper only lays the foundation of a new concept and there are still many questions yet to be answered. First, it would be interesting to determine what is the best position for adding dummy buffer in the source code, in order to get the best reliability. Second, clearly allocating more memory space in a program may lead to overheads. The question is how to get the best balance between security and performance when using this technique. Third, currently, the approach is for handling memory-related attacks, can the method be extended to different types of attacks? Addressing these issues is our immediate next research step.

9. REFERENCES

[1] M. G. Andrea Cugliari. *Smashing the stack in 2010*. PhD thesis, Politecnico di Torino, 2010.

[2] G. Antoniol. Search based software testing for software security: Breaking code to make it safer. In *Proc. of the IEEE International Conference on Software Testing Verification and Validation Workshops ICSTW 09*, 2009.

[3] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11:1491–1501, December 1985.

[4] A. Avizienis. The methodology of n-version programming. In *Proc. of Software Fault Tolerance*, pages 23–46, 1995.

[5] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[6] S. Bhatkar. *Defeating memory error exploits using automated software diversity*. PhD thesis, NY, USA, 2007. AAI3337612.

[7] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 17–17, 2005.

[8] K. P. Birman and F. B. Schneider. The monoculture risk put into context. *IEEE Security and Privacy*, 7(1):14–17, 2009.

[9] blexim. Basic integer overflows. http://www.phrack.org/issues.html?issue=60&id=10#article.

[10] S. S. Brilliant, J. C. Knight, and N. G. Leveson. The consistent comparison problem in n-version software. *IEEE Transactions on Software Engineering*, 15:1481–1485, 1989.

[11] c0ntext. Bypassing non-executable-stack during exploitation using return-to-libc. http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard : Protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*, 2003.

[13] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[14] S. Fitzpatrick, C. Green, S. Westfold, J. McDonald, and A. Coglio. Using software generation and for cyber-defense. In *Survivability in Cyberspace*, 2011.

[15] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. of the 10th ACM Conference on Computer and Communications Security*, pages 272–280, NY, USA, 2003.

[16] F. Lindner. A heap of risk. http://www.h-online.com/security/features/A-Heap-of-Risk-747161.html, 2006.

[17] M. Milenković, A. Milenković, and E. Jovanov. Using instruction block signatures to counter code injection attacks. *SIGARCH Comput. Archit. News*, 33:108–117, 2005.

[18] T. S. Perraju, S. P. Rana, and S. P. Sarkar. Specifying fault tolerance in mission critical systems. In *Proc. of the 1996 High-Assurance Systems Engineering Workshop*, HASE '96, pages 24–30, Washington, DC, USA, 1996. IEEE Computer Society.

[19] G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Procceedings of the 26th Annual Computer Security Applications Conference*, pages 41–48, NY, USA, 2010.

[20] B. L. C. Ramos. Challenging malicious inputs with fault tolerance techniques. https://www.blackhat.com/presentations/bh-europe-07/Luiz_Ramos/Whitepaper/bh-eu-07-luiz_ramos-WP.pdf, 2007.

[21] R. J. Rodriguez and J. Merseguer. Integrating Fault-Tolerant Techniques into the Design of Critical Systems. In *ISARCS'10: Proc. of the 1st International Symposium on Architecting Critical Systems*, volume 6150 of *Lecture Notes in Computer Science*, pages 33–51. Springer, 2010.

[22] W. Schwartau. *Time-Based Security: Practical and Provable Methods to Protect Enterprise and Infrastructure, Networks and Nation*. Interpact Press, 1999.

[23] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, 2004.

[24] P. Silberman and R. Johnson. A comparison of buffer overflow prevention implementations and weaknesses. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf, 2004.

[25] K. suk Lhee and S. J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33:423–460, 2002.

[26] T. P. Team. Documentation of the pax project. http://pax.grsecurity.net/docs/.

[27] P. Wagle and C. Cowan. Stackguard: Simple stack smash protection for gcc. In *Proc. of the GCC Developers Summit*, pages 243–255, 2003.

[28] D. W. Williams, W. Hu, J. W. Davidson, J. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security & Privacy*, 7(1):26–33, 2009.