DAWSON -- Synthetic Diversity for Intrusion Tolerance

Global Infotek, Inc.: James Just, Nathan Li
UC Davis: Karl Levitt, Jeff Rowe, Tufan Demire
SUNY-Stonybrook: R. Sekar

1. Introduction

Fault-tolerant and intrusion-tolerant technologies can be used to implement robust systems that adjust to accidental and deliberate faults. Such systems have an Achilles heel since their operation depends upon failover to healthy spares. Continued attacks can deplete the supply of usable spares until the system is no longer capable of delivering the minimum required functionality. This is a particular problem for large computer monocultures like the Wintel platform. A serious related problem of monocultures is the risk of common mode vulnerabilities that lead to exponentially cascading failures (e.g., the Slammer worm epidemic).

The DAWSON (Diversity Algorithms for Worrisome Software and Networks) project is part of the DARPA Self Regenerative System (SRS) program. It is developing mechanisms that mitigate both the problems of spares for commercial off-the-shelf (COTS) software and of common mode failures in computer monocultures. DARPA has established a goal for SRS projects in the diversity area to meet. That goal is generating 100 variants of any original program with no more than 30% of the variants being vulnerable to the same exploit as the original program. DAWSON’s internal goal is to beat that metric by at least a factor of ten. As we show below, we have substantially exceeded both the program goal and our internal goal.

2. Breaking Vulnerability Specifications

Remote computer attacks depend upon very specific details of executing machine code, e.g., relative or absolute addresses of vulnerable pointers or critical data values. Attackers inject their own code or misuse existing code by exploiting these details and then go on to find and access system calls to read files, write files, propagate, leak important information, etc. The machine code details are essentially fixed in computer monocultures because of how code is written and compiled and how operating systems link and load executable programs. We refer to these details as the Vulnerability Specification or V-SPEC that attackers count on. As illustrated on the next page, any executing code that supports a given V-SPEC can be exploited by the identical attack.

DAWSON transforms programs in ways that break the V-SPEC but don’t affect the functional behavior of a program. It introduces synthetic diversity at the executable code level by randomizing selected aspects of the program address space so that injected code no longer functions and existing code is no longer reachable.

---

1 This work was partially funded by Defense Advanced Research Project Agency under contract # FA8750-04-C-0244. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Project Agency or the U.S. Government
3. DAWSON Architecture

The ease of transforming code without breaking it diminishes as source code is compiled into object code, linked into executable code, and loaded into memory as machine code. Some transforms are easy and safe to perform at the executable code level (e.g., on disk or in memory). Other transforms require analysis of disassembled binary code to prevent breakage. Still other transforms can only be accomplished at the source code level. More sophisticated transforms can be made on executable code if relocation information about that executable is available.

DAWSON’ functional architecture accommodates these differences (see below). A transformed program executes on the same hardware but has a distinctly different address space and interface with its environment. Every time a DAWSON-protected application is loaded, it is transformed

Diversity System Functional Architecture

Address randomization does not remove vulnerability but makes effect of attack unpredictable

*Pseudo-Random Number Generator
differently based upon a pseudo-random number. The optional annotation file contains relocation information for a program and can be produced by an offline computer or human process.

In a DAWSON-protected system, most attacks will fail because the assumed location of the vulnerability is different. Others will fail because the injected commands do not find the system calls or resource names they need. With high probability, attacking code will simply fail and crash the process or endlessly loop (which increases the likelihood of detection).

4. Implementation

DAWSON contains three levels of defense. The first level is designed to "break the exploit specification" and make the initial exploit of any vulnerability much more difficult to achieve on a wide scale. If the initial exploit does succeed, the second level of defense makes any subsequent activity by the injected code (which may now control the process) much more likely to crash the system than perform its intended mission. We call this “breaking the payload specification”. The third layer completes the defense in depth by “preventing by-pass of protection mechanisms” and setting various traps and tripwires that alert on attempted intrusions. We have made good progress on all fronts and have initiated an extensive evaluation program that includes analysis, experimental testing, and Red Teaming. We are also looking at approaches are facilitated by advanced compiler and instrumentation technologies.

Various low-level operating system mechanisms (such as hooking selected function calls, injecting custom DLLs into running processes, and loading special kernel mode drivers) are used to implement these defensive layers and these are described in more detail below.

4.1 Breaking the Exploit

Executable Base Address Randomization

DAWSON is able to relocate essentially everything except the initial thread of the main program. Main programs of executables do not require relocation information since they are the first modules to be loaded and cannot conflict with another module. How much risk remains if the main thread cannot be randomized is uncertain for most commercial programs that are heavily multi-threaded. Nevertheless, we are working on mechanism to randomize main programs of executables. Some compilers include relocation information for executables in the generated PE files. These executables can be randomized.

Another approach is to recompile the executable as a relocatable DLL and apply the mechanisms described in section 1.2. We are also working on approaches that extract relocation information from disassembled binary code with and without symbol table information. A kernel mode driver

DLL Base Address Randomization

The Windows loader is responsible for loading any required library (DLL) into a user process and returning its handle. The loader first tries to put the library at its default address (as specified in PE header) or, if that address is not available, into the first free address. Thus libraries usually load at the same address in a given application -- an invariant that is easy to exploit. DAWSON hooks two function calls (LdrpMapDLL and NtMapViewOfSection) that are responsible for DLL address mapping and randomizes the default address during loading.
This works for all DLLs except ntdll.dll and kernel32.dll which are loaded into fixed locations very early in the process. The approach for randomizing the location of these two DLLs is described under DLL Rebasing. Note that randomizing these two DLLs is part of layer 1 as well as layer 2. In layer 1, its role is to defeat existing code attacks. In layer 2, its role is to prevent injected code from using system API.

**Stack Randomization:**

Stack base randomization is applied to all but the initial thread and involves two steps:

*Stack Location Randomization:* After identifying the source that is responsible for creating stack space and hooking the function to provide the randomized address, this randomized address can be anywhere in the 2GB user address space, as long as the address is free and the stack location will be greatly different each time.

*Stack Startup Frame Randomization:* After the stack space is randomized in the user address space, within each stack space, we can randomize the location where the thread start routine started by hooking the `CreateThread` function. By inserting a fake `Thread_START_ROUTINE` and allocating random-sized space from the stack, we effectively can move the real start routine by randomized distance from its original location.

The primary thread can’t be randomized this way. Although, in a practical sense, the primary thread is less prone to attack, we can use a different approach to randomize the base of the initial stack. It involves inserting a new segment into the PE file and modifying the main program.

**Heap Randomization:**

DAWSON hooks the APIs in NTDLL that manage the heap (`RtlCreateHeap, RtlAllocateHeap, RtlReAllocate, RtlFreeHeap, RtlSizeHeap, RtlDestroyHeap`). This enables us to randomize all heap allocation, reallocation and free activities without modifying other code. The heap randomization is different for each invocation and includes:

- Heap base
- Heap block distance
- Heap block random-sized header and tail
- Randomized magic number embedded in header and tail

**4.2 Breaking the Payload**

We have developed two methods for denying an attacker’s injected code access to critical DLLs. We have successfully relocated two critical libraries (kernel32.dll and ntdll.dll) that are typically loaded in the same small memory range. We have also successfully masked the Windows Process Environment Block, which has a wealth of information useful to the attacker.

We are also working on locking down the exception handling process and identifying other sources of information useful to attackers.

*kernel32 and ntdll Rebasing*
Rebasing these two DLLs is cumbersome but we are working to improve the process. Any DLL being used by an application cannot be rebased and these two DLLs are used by all applications. Thus we rebase a copy of them in Safe mode and use the MoveFileEx function with a flag (MOVEFILE_DELAY_UNTIL_REBOOT) set in the registry to rename the file after a reboot into normal mode. This is obviously not as easy as restarting an application.

**PEB Masking**

The process environment block (PEB) is a high-level user mode structure that contains some important information about the current process. It’s a quite lengthy structure, it has around 70 fields. This structure is not well documented and is described in detail usually by reverse engineers. PEB contains very useful information for the process and also for the attacker. For example, PEB has a field named ProcessModuleInfo. This field is of type PEB_LDR_DATA (loader data). The loader data structure contains all the information about the base addresses and names of the DLLs and the executable that is in memory. The attacker can easily scan the PEB structure to find out the base address of important DLLs, because PEB is at a fixed place in memory. MS Blaster worm [9] is known to use this structure during the attack.

After analyzing ntdll.dll with a disassembler, we can see that PEB is accessed through the thread environment block (TEB) data structure in the following way:

```
Mov eax, fs:18h // Access TEB using fs register
Mov eax, [eax+30h] // Access PEB, which is pointed by TEB
Mov eax, [eax+0Ch] // Access ProcessModuleInfo, pointed by PEB
```

The loader data is pointed by ProcessModuleInfo and contains three pointers to three linked lists, which are load order module list, memory order module list and initialization order module list. Each element in the linked list contains information about the loaded modules, such as base address, entry point, size, DLL name etc.

The loader data is not accessed very frequently once the process’s memory is initialized. To prevent the attacker from accessing the list of loaded modules, we make changes to the PEB, such as encrypting some important pointers. Also, normal programs have to work as before, therefore we need to change the functions that access the PEB. Again we use a wrapper function for this purpose. The wrapper decrypts the pointers, calls the original function that accesses the PEB, and then encrypts the relevant pointers again. This way, the important information remains unreachable by the attacker. Fortunately, the number of functions that access the loader data seems to be small. Such functions are LoadLibrary, GetProcAddress, GetModuleHandle, FreeLibrary, etc. All of these functions are in user space.

**4.3 Non-Bypassability, Traps and Tripwires**

With regard to non-bypassability within DAWSON’s third layer of defense, our design includes the use of wrapper technology previously developed under DARPA sponsorship. This technology mediates the kernel’s callgate interface to prevent bypass. This is part of the third layer of defense in DAWSON architecture.

We are also investigating non-bypassable technology being developed under other auspices by UC Davis. Their technology replaces the system call ID in the EAX register with a signed combination of the return address and the system call id and then uses a kernel device driver to
validate the origin of system calls. If this technology proves out, it will be considered as an alternative implementation for the design. Non-bypassability is not being implemented under current project funding.

DAWSON is implementing measures in its level 3 defenses that make it more difficult for an attacker to mount brute force or intelligent derandomizing attacks. By marking blocks of memory as non-readable and locating exception generators at known vulnerable addresses, we are able to establish traps and tripwires for the unwary attacker.

4.4 Advanced Compiler and Instrumentation Technology

UC Davis has access to Microsoft’s prototype Phoenix suite of tools. Phoenix is supposed to be the basis for all future generation Microsoft compilers. Vulcan is the binary editing facility of Phoenix. It utilizes a large database of program information that is generated by the new compilers to provide the capability to statically modify application binaries and dynamically transform running applications. Vulcan handles all the relative and absolute address computations so that dead code can be inserted and basic code blocks and static data can be randomized in the executable without breaking it.

While this technology is proprietary and unavailable commercially, our study indicates enormous potential for introducing diversity into executables that have the associated program database.

4.5 Testing Effectiveness Relative to Program Metrics

DAWSON has a comprehensive approach to measuring effectiveness relative to the SRS program’s diversity metric of producing 100 variants of the original program with no more than 30% vulnerable to the same exploit as the original program. The approach includes theoretical assessments of the effort required to defeat our randomization implementations (related to the size of the randomization space), extensive lab testing of our mechanisms against known and zero-day advanced attacks in real and synthetic applications, and participation in an upcoming Red Team exercise to evaluate advanced derandomization attacks and by-pass attacks. We are also evaluating models of memory error attacks to assess the comprehensiveness of our approach.

5. Results and Conclusions to Date

Theoretical analysis of the DAWSON mechanisms for absolute address randomization indicates that they are effective against all attacks reported so far and likely for most future attacks as well. Some residual vulnerabilities exist, but these are mainly theoretical possibilities at the moment. We have calculated the exploit effort for our mechanisms. Exploit effort is a measure of the size of the randomization space and an indicator of the number of variants that can be generated without duplication of vulnerabilities). It ranges between 13K and 5M for different attacks. In the future, we hope to improve this to between 500K and 50M.

We have implemented a 100 node testbed in Emulab at the University of Utah using their newly released capability support Windows XP SP1 and SP2 hosts. We are using it to evaluate the effectiveness of our diversity mechanisms on a larger scale of testing than is possible in our laboratory. We have installed both real applications and our own synthetic service on each to test
the effectiveness and performance impacts of the DAWSON mechanisms. We implemented the synthetic service so that it contains a number of advanced memory error vulnerabilities. We also installed our Catalyst test harness on each host and used Experiment Monitoring Display to gather and display results during the test.

We also developed associated exploits for the new vulnerabilities and installed these exploits in the open source Metasploit framework. We implemented a new payload that writes a simple marker rather than returning a command window and installed it in Metasploit also. We then automated Metasploit so that we could run a series of tests with representative exploits that cover a subset of the spectrum of memory errors from traditional stack smashing attacks to more advanced heap overflow, return to libc, and format string attacks. The automation also fed data to the Experiment Monitoring Display for completeness and reset the attacked hosts after each trial. The defense mechanisms generated a new version of randomization for each trial and each host was randomized differently on each trial.

Our initial empirical results were generated using mostly layer one defense mechanisms. They support the analytic results that show very low probabilities of successful attacks. When all of the defense mechanisms are enabled, no attack in the repertoire succeeds. Since this is a boring graphic, we have not included it. A more interesting chart is the one below that show the results of running a given attack, specifically the heap overflow attack, many times against the system with only a single different defense mechanism enabled for each trial. The attack generally succeeds against all the individual defense mechanisms tested except for the heap overflow protection mechanism. While this may seem like an obvious result, it is not. We will be testing each of our mechanisms against each of our attacks to characterize the effectiveness of our defense mechanisms against different types of attacks.

**Evaluation Results**

**Experiment July 10**

![Host Attack Status](image)

**Attack Type: Heap Overflow**
- **Trial 4:** No Randomization: Attack is successful 98/100 times
- **Trial 5:** Stack Randomization: Attack is successful every time
- **Trial 6: Heap Randomization:** 0 successful attacks
- **Trial 7:** IAT Randomization: Attack is successful 99/100 times

The next chart summarizes the performance impact along several dimensions for some of our defense mechanisms. The dimensions include changes in the file size on disk, changes in the memory footprint, changes in the load time, and changes in the run time of each randomized
application relative to the non-randomized version. As shown, the impacts are minimal and, in all cases, are less than 5%. We will begin testing with a Red Team in the near future.

Performance Impact

<table>
<thead>
<tr>
<th>Defense Technology</th>
<th>Disk File Size</th>
<th>Memory Usage</th>
<th>Load Time Increase</th>
<th>Run Time Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLL Base Randomization</td>
<td>NA</td>
<td>NA</td>
<td>&lt; 1 millisec</td>
<td>NA</td>
</tr>
<tr>
<td>Stack Randomization</td>
<td>NA</td>
<td>NA</td>
<td>&lt; 1 millisec</td>
<td>NA</td>
</tr>
<tr>
<td>Heap Base Randomization</td>
<td>NA</td>
<td>NA</td>
<td>&lt; 1 millisec</td>
<td>NA</td>
</tr>
<tr>
<td>Heap Block Randomization</td>
<td>NA</td>
<td>Up to 16 bytes per block</td>
<td>NA</td>
<td>&lt; 5%</td>
</tr>
</tbody>
</table>

* Data collected on a Pentium 4 1.2GHz CPU with 768MB RAM

These initial results suggest that DAWSON provides extremely effective and low cost protection against attacks that exploit common mode vulnerabilities in the Windows monoculture.