Dynamic Code Instrumentation to Detect and Recover from Return Address Corruption

Suhas Gupta  
Computer Sc. Dept.  
IIT, Delhi  
India  
suhas@cse.iitd.ac.in

Pranay Pratap  
Computer Sc. Dept.  
IIT, Delhi  
India  
pranaypratap@gmail.com

Huzur Saran  
Computer Sc. Dept.  
IIT, Delhi  
India  
saran@cse.iitd.ac.in

S.Arun-Kumar  
Computer Sc. Dept.  
IIT, Delhi  
India  
sak@cse.iitd.ac.in

ABSTRACT

Return address corruption on the stack using buffer overflow attacks is one of the most common ways in which the security of a system can be compromised. This paper provides a way of detecting return address corruption on the stack using dynamic code instrumentation. The detection is done at run-time and it does not depend on the availability of source code of the vulnerable application. The approach we are presenting is not limited only to buffer overflows, rather it can handle any kind of return address corruption. Furthermore, cases in which recovery from stack corruption is possible and the mechanisms for recovery in such cases have also been discussed.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—Code inspections and walk-throughs, Error handling and recovery

General Terms

Security, Experimentation

Keywords

Dynamic Code Instrumentation, Buffer Overflows, Return Address Corruption

1. INTRODUCTION

In programming languages like C, it is possible to corrupt the execution stack by writing past the end of an array declared in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. The “stack smashing”[10] technique is the most common way used in exploits to break the security of programs. It is based on sending an unexpected amount of input data to a program causing a buffer overflow that allows the attacker to make the program execute arbitrary assembler code which can grant to him the access to the system, destroy the system files or do anything else.

The stack can also be smashed through format string vulnerabilities and return-to-libc attacks[9]. In all these cases, the return address is overwritten, disrupting the normal course of execution of the program and executing malicious code instead.

There are many ways to prevent stack smashing attacks. Static approaches though faster and easier have many pitfalls. Most static approaches can be bypassed if the attacker knows exactly what is being done to secure the stack. For example, there are known ways for circumventing StackGuard and Stack Shield[5]. Moreover, static approaches need to compile the code or have to be integrated into the compiler. So a static approach has to be incorporated into the code development process. A dynamic approach will need only binaries and does not depend on the availability of the source code. A dynamic approach may also help in making smart decisions at runtime. For platforms like Windows, where source code is seldom available, this is usually the only feasible solution. Moreover, most security issues are relevant to the Windows platform.

Our goal was to develop a dynamic code instrumentation framework which could effectively detect return address corruption. In Section 2, we discuss existing approaches for securing the stack. In Section 3, we describe the basic instrumentation framework. Section 4 focuses on using this framework to detect return address corruption. Recovery possibilities are explored in Section 5. Section 6 discusses the application of the framework in a practical scenario. In Section 7, we illustrate the performance impact of our framework. Finally we discuss the shortcomings and future directions in which this approach can be extended.

2. RELATED WORK

There are several approaches that have been explored in the past to counter the stack smashing problem. They can be broadly classified into:

- static techniques at compile time
- dynamic instrumentation
- binary modification
- disabling stack execution
- safer C library support etc.
Static techniques involve embedding inbuilt checks into code when it is being compiled. Such checks can include range checking or return address protection using a canary etc. Stack Shield[1] is a tool for adding protection to programs at compile time without changing a line of code. The Stack Shield protection system copies the return address in a non-overflowable location on function prologue and checks if the two values are different on function epilogue. If the two values are different, the return address has been modified so Stack Shield terminates the program or tries to let the program run ignoring the attack (risking at maximum a program crash). StackGuard[6] is another tool that detects and defeats stack smashing attacks using compiler techniques. StackGuard places a "canary" word next to the return address when a function is called. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted, and the program responds by emitting an intruder alert into syslog, and then halts.

Dynamic instrumentation techniques involve binary rewriting in memory at runtime in order to incorporate checks and security measures. Strata[13] is a Software Dynamic Translation (SDT) infrastructure which interposes a software layer between the application and the CPU, much like the virtual machine model. This SDT infrastructure can be extended to prevent stack smashing attacks as discussed in [13]. DynamoRIO[8] is another dynamic code modification system which is based on Dynamo[3] (Dynamic Optimizer) from Hewlett Packard Laboratories. DynamoRIO supports code transformations on any part of a program. It exports an interface for building dynamic tools for a wide variety of uses: program analysis and understanding, profiling, instrumentation, optimization, translation, etc. DynamoRIO shifts an application’s execution from its original instructions to a code cache, where the instructions are then modified. DynamoRIO then forces the code within its code cache to adhere to custom rules. Detours[7] and PIN[12] are also dynamic instrumentation tools which can possibly be used for securing the stack. The libverify[4] library relies on verification of a function’s return address, similar to StackGuard[6]. In contrast to StackGuard, libverify adds verification code at runtime by modifying process memory. Conceptually, the framework we have developed is closest to libverify.

Static binary modification involves static instrumentation of executable files to add checks which prevent buffer overflow. RAD[11] is one such effort. It uses a binary rewriting approach to augment existing Portable Executable (PE) binaries with a return address defense (RAD) mechanism, which protects the integrity of the return address on the stack by keeping a redundant copy.

Most systems do not need code to be ever executed on the stack. Since the most common buffer overflows, rely on code to be injected into the buffer and then executed, another solution is the option to install the operating system with stack execution disabled.

Another anti stack smashing strategy would be to provide a safe version to the C library functions on which the attack relies to overwrite the return address. This can be achieved by dynamically instrumenting these or just providing alternative safe functions for these or by permanently replacing the vulnerable functions with safer ones. libsafe[4] is one such effort.

3. INSTRUMENTATION FRAMEWORK

Our approach is to replace the first few instructions in a function by a jump to another section of code which is allocated separately. A relative jmp being 5 bytes, the instructions replaced should atleast be 5 bytes in size. The extra bytes are replaced by nops. The new section (prologue) contains the replaced instructions of the function along with our custom code. At the end of the new custom code section, a jump is placed to return to the appropriate instruction in the original function. The same needs to be done at the end of the function where the last few instructions are replaced by a jump and this jump again points to our custom code (epilogue) which is also allocated dynamically. In this case we do not jump back to the original function code since the last instruction in the custom code is the copied return instruction, which anyway ends the function call. The framework is illustrated in Fig 1.

4. DETECTION OF RETURN ADDRESS CORRUPTION

The basic idea is to store a copy of the return address at the beginning of the function and cross check the current return address with the previously stored value before function exit. If the values match, then it can be safely assumed that the stack was not corrupted and normal execution continues. Otherwise, some suitable action as required may be taken. The strength of this approach lies in its ability to catch all types of return address corruptions and not only those caused by buffer overflows.

Also, unlike StackGuard which pushes the canary on the original stack itself, we maintain a separate stack on which we push a copy of the return address every time an instrumented function is called and we pop the value when this function returns. This auxiliary stack is allocated and maintained within the process space of the instrumented process. Thus, we avoid the potential risks of pushing the return address(canary value in the case of StackGuard) on the same stack. Since the stored value is on a separate stack, it is not possible that the stored value could also be overwritten in such a manner that the buffer overflow would go undetected.

With reference to the generic instrumentation framework,
the prologue will contain code which saves the return address separately. Similarly, epilogue contains the code for cross checking the current return address on the stack with the previously saved value.

\texttt{foo():}
\begin{verbatim}
  jmp prologue
L1: foo() code
  jmp epilogue
end prologue
\end{verbatim}

\texttt{prologue (memory address):}
\begin{enumerate}
  \item backup the registers we want to use
  \item save return address on parallel stack
  \item first bytes of foo which were replaced by jmp
  \item jmp to L1
\end{enumerate}

\texttt{epilogue (memory address):}
\begin{enumerate}
  \item backup the registers we want to use
  \item cross-check the return address
  \item continue to L2 if ret addr valid
  \item else
    \begin{enumerate}
      \item take suitable action
    \end{enumerate}
L2: last bytes of foo which were replaced by jmp
\end{enumerate}

5. RECOVERY FROM RETURN ADDRESS CORRUPTION

Detection of return address corruption provides means to thwart any attempt to smash the stack. In an ideal scenario recovery from an attack would be the ultimate goal and not necessarily letting the application just crash. Recovering from a buffer overflow is not easy because the stack may already have been irreparably corrupted. It must be understood that recovery in all cases will not be possible.

One heuristic which can be applied to solve this issue is to save the previous stack state before each function call and if an overflow occurs we may restore the corrupted stack with the stored stack state. This can be achieved by copying the contents of the stack onto a parallel stack. We are maintaining a backup stack separately and it is updated just before each function call and again just before each valid return. The stack state is restored in case return address corruption is detected.

Instead of backing up the whole stack we just keep a copy of the penultimate stack frame as backup. This can be extended to a fixed number of stack frames if required.

5.1 Legitimate Return without Stack Recovery

The detection of return address corruption requires us to store the return address on a parallel stack. This stored return address is needed to cross-check the address on the stack during corruption detection. We can use this stored backup value to restore the actual return address on the stack and let the code execution continue normally. This will ensure that the code jumps to the correct instruction after the current function returns. This strategy helps maintain the legal execution sequence of the application.

This approach will work in only the most trivial cases for application recovery. In cases where the stack has been corrupted beyond the current stack frame also corrupting the previous frame on the stack, the application will most probably crash at a later stage. However, this will still not transfer the control to the attacker and the worst case scenario would be an application crash.

5.2 Legitimate Return with Stack Recovery

A logical extension to the previous approach would be to restore the stack state before returning to the correct address. Ideally one would like to restrict the corruption to the current frame but owing to the difficulty in doing so we instead restore the penultimate stack frame. We copy the previous stack frame to a parallel stack before the function call and if we detect an overflow, we restore the previous stack and then return to the correct address (Fig 2).

For this we need to modify the function prologue and epilogue we add to each function we instrument.

\texttt{foo():}
\begin{verbatim}
  jmp prologue
L1: foo() code
  jmp epilogue
end prologue
\end{verbatim}

\texttt{prologue (memory address):}
\begin{enumerate}
  \item backup the registers we want to use
  \item save return address on parallel stack
  \item copy previous frame
  \item first bytes of foo which were replaced by jmp
  \item jmp to L1
\end{enumerate}

\texttt{epilogue (memory address):}
\begin{enumerate}
  \item backup the registers we want to use
  \item cross-check the return address
  \item continue to L2 if ret addr valid
  \item else
    \begin{enumerate}
      \item restore previous stack frame from backup
      \item restore the return address
    \end{enumerate}
L2: last bytes of foo which were replaced by jmp
\end{enumerate}

This solution will work if we can be sure that the corruption has spread to one penultimate frame only. In case more than one penultimate frame has been corrupted, we cannot use this mechanism to recover after a stack smashing attack. The code will most probably crash at some later stage due to the corruption on the stack.
Also, in case of function arguments being passed by reference, then just restoring the stack might change the variable fields (the function arguments point to) back to their original value instead of what the current function modified them to. This scenario will also crash the code at a later stage. We need to check if the current function has some of its arguments being passed by reference and if so, it will probably be better to just terminate the application than trying to recover from the crash.

5.3 Repeat Function Call

Another option is to restore the previous stack frame and then call the current function again. For this we needed to store the function arguments as well when we save the stack state. After detecting return address corruption, we recover the previous stack frame and also the function arguments of the current function. Instead of returning from this function, we restore the stack to the state before this function was called. Then we call this function again with the same arguments (Fig 3). The idea behind this approach is to try another time and if the return address is not corrupted this time, we simply let the execution continue. In case corruption is detected again, we terminate the application.

This approach will work in cases when the erroneous data that has caused the overflow is not static (input dependent) or not in a buffer. In such cases the next execution of the function might run safely but if the data is indeed persistent then the function is doomed to crash again and subsequent return address corruption should lead to application termination.

\[
\begin{align*}
\text{foo():} & \quad \text{jmp prologue} \\
& \quad \text{L1: foo() code} \quad \text{jmp epilogue}
\end{align*}
\]

\[
\begin{align*}
\text{prologue (memory address):} & \quad \text{backup the registers we want to use} \\
& \quad \text{save return address on parallel stack} \\
& \quad \text{copy previous frame} \\
& \quad \text{first bytes of foo which were replaced by jmp} \\
& \quad \text{jmp to L1}
\end{align*}
\]

\[
\begin{align*}
\text{epilogue (memory address):} & \quad \text{backup the registers we want to use} \\
& \quad \text{cross-check the return address} \\
& \quad \text{continue to L2 if ret addr valid} \\
& \quad \text{else} \\
& \quad \text{restore previous stack frame from backup} \\
& \quad \text{restore saved hello arguments on the stack} \\
& \quad \text{restore the return address} \\
& \quad \text{jump to foo() again}
\end{align*}
\]

L2: last bytes of foo which were replaced by jmp

6. JUST IN TIME PROTECTION

The instrumentation framework discussed above has been successfully extended as a practical solution to a known problem. An overflow happens in Internet Explorer’s HTTP parsing code which is triggered when it receives an HTTP reply with excessively long values in both “Content-type” and “Content-encoding” fields[2]. The vulnerable code resides in urlmon.dll. By exploiting this vulnerability, an attacker can run arbitrary code on a victim’s computer.

We successfully instrumented the vulnerable function in the process space of a running instance of IExplore.exe (in urlmon.dll) and prevented a buffer overflow attack from taking control of the target system. Moreover, we were able to pinpoint the exact packet which was the source of the malicious data and thus, providing the user with a choice to block the IP/port from which the packet originated. To achieve this end another thread is spawned as the process (Internet Explorer) starts executing. The latter constantly monitors the network traffic and captures the last \( n \) packets (\( n \) can be as large as required, we used \( n = 5000 \)). The captured packets are stored on disk in a file. When a corruption is detected on Internet Explorer’s stack (specifically return address corruption in the vulnerable function that we instrumented), the packet capture thread is notified of the corrupt address. The packet capture thread then stops capturing the packets and scans the already captured packets for the corrupt return address value from which the offending packet is singled out, the IP address from which it originated, the port on which it was received are determined. The packet capturing thread may take appropriate action.

7. PERFORMANCE IMPACT

Any dynamic code instrumentation framework must be developed keeping the performance impact in perspective. Traditionally security (in the form of checks built into the code) has been sacrificed to improve performance.

The performance impact of “return address corruption” detection can be measured in terms of two factors:

1. Cost of instrumenting a function
2. Cost of execution of an instrumented function call vs a normal function call

7.1 Cost of Instrumenting a Function

Fig 4 shows the behaviour of the cost of instrumenting a function as the function size increases. It is not possible to instrument the function alone and make subsequent instrumentation calls on the function address. Once the
Figure 4: Graph for Instrumentation-Deinstrumentation Cost

Figure 5: Performance Impact of Instrumentation on Function Execution
function is instrumented an attempt to instrument it again might result in a crash and also add to the overhead which will give us inconsistent results for our statistical analysis. So, we added a deinstrumentation routine to deinstrument a function after each instrumentation. This cycle was run a number of times to get an idea of the cost of the instrumentation routines. The instrumentation cost turns out to be linearly dependent on the function size. This can be explained by the fact that a full traversal of the function code is required to find its exit point. Thus, increase in function size leads to more time being spent in the traversal.

7.2 Cost of Execution of an Instrumented Function Call vs a Normal Function Call

The graph in Fig 5 shows the instrumentation overhead when executing functions of different sizes. As expected, the instrumentation overhead is constant and independent of the function size. This is because the framework adds only a fixed number of instructions to each function irrespective of its size. The extra time required to execute these instructions is a constant. This constant overhead was approximately $0.25 \times 10^{-7}$ sec per function call for the testing done on a P-IV(HT) 3.2 GHz machine with 1 GB RAM.

8. FUTURE WORK

We have developed a framework which can help us detect and recover from return address corruption scenarios in most cases. The logical extension of this framework would be to extend this framework so that detection works for all of the cases and recovery where possible.

1. **Gauge the Extent of Corruption:** Attempts should be made to gauge the extent of corruption on the stack. This can be achieved by figuring out ways to save return addresses for previous frames also. It is not necessary to store the return addresses only; any information for the top few stack frames which can later be crosschecked and verified to help us determine whether the stack frame in question is corrupted or not, is good enough. This can help us to determine if the stack has been corrupted at various levels and come to a conclusion about the level of corruption. This will help us in determining whether recovery is possible at all and if so, how to proceed with it.

2. **Save Multiple Stack Frames for Later Recovery:** It is important that we are able to recover even if the corruption has spread beyond the current stack frame if possible. This is not an easy task as gauging the depth of corruption is not going to be easy. Moreover, making full recovery after such an attack might not even be possible. If we save multiple stack frames for later recovery, we might be able to recover the damage by restoring the top few corrupted frames. This approach will depend on our ability to find the depth of corruption and whether any argument passed to one of the corrupted stack frames were by reference because in this case we will not be able to restore the stack to a consistent state.

3. **Handle Arguments being Passed by Reference:** Arguments passed by reference pose a problem to our recovery module as restoring the stack with the saved stack frames might result in an inconsistent state of clearpage the stack. We might copy back a penultimate frame such that the arguments that were passed by reference were reverted back to their original values, when they should have been updated by the current function call. One approach to counter this would be to store the addresses these reference variables point to when saving the stack. Then verify whether they are still pointing to the same addresses before returning. Saving these addresses properly can help us decide whether to revert back their values during recovery or that we should actually leave their current values as it is.

9. CONCLUSION

The framework that we have developed can detect return address corruption in every scenario. Moreover, it even makes recovery (after corruption) possible in some cases. Most other solutions for stack security such as StackGuard[6], Stack Shield[1], libverify[4], DynamoRio[8] etc try detection of return address corruption but this paper also explores scenarios in which recovery is possible. Furthermore, unlike StackGuard and Stack Shield our detection module cannot be bypassed because we dynamically allocate the memory where a copy of the return address is stored. Our framework has an advantage over RAD[11] too because dynamic instrumentation allows us to cover all client code unlike RAD which might fail while disassembling binaries if code and data segments are mixed or indirect branches are used[11]. Another advantage our framework has is that it can be used to selectively instrument vulnerable functions on a per process basis as compared to RAD which instruments whole binaries. libverify also instruments the whole process. By selectively instrumenting the vulnerable functions, we are avoiding the extra overhead incurred by RAD and libverify. The instrumentation framework we have developed is generic enough to be extended to instrument functions selectively for other purposes also. For instance this framework can be extended to do code profiling and even prevention of buffer overflows in case of known vulnerabilities.

10. REFERENCES


