matasano

Secure 64 SourceT OS Security Evaluation Report

Matasano Security, LLC
November 20, 2007

Table of Contents

1. Executive Summary 1

   1.1 Introduction 1

   1.2 Summary of Findings 1

2. Methodology 2

3. Overview of Secure64 Technology 2

4. Areas of Analysis 3

   4.1 Code Injection 3

      4.1.1 Traditional Application Model Code Injection 3

      4.1.2 Secure64 SourceT Application Code Injection 9

   4.2 Privilege escalation 17

      4.2.1 Traditional model privilege escalation 18

      4.2.2 Secure64 SourceT Privilege Escalation 19

   4.3 Boot Process 21

      4.3.1 Traditional Boot process 21

      4.3.2 Secure64 SourceT Boot process 22

5. Conclusions 24

6. Matasano Background 25

   6.1 Corporate Overview 25

   6.2 Locations 26

1. Executive Summary

1.1 Introduction

Matasano was tasked by Secure64 to evaluate critically the claim that, for remote attackers, SourceT is "immune to all forms of malware, including rootkits, Trojans, viruses and worms." To do this, Matasano evaluated SourceT against three areas of vulnerability: code injection, privilege level escalation, and alteration or subversion of the trusted boot process. These areas were selected as they comprise the strategy of typical malware such as worms, spyware or trojan horse applications to introduce arbitrary code into a computing system.

1.2 Summary of Findings

No method for the injection of arbitrary code was identified in architectural analysis. By eliminating any location an attacker can write executable code, it becomes very difficult for any traditional form of attack to result in controllable execution by an attacker. While the potential exists for exploits, they are of a decidedly application specific nature, depending on the presence of certain elements arranged in a particular manner, such as function pointers that can be overwritten via user supplied data, and cause the execution of functions *already present* in the application that would grant some level of programatic control. Such a scenario, while possible, can be avoided through sound application architecture and design methodology.

If an attacker was successful in inducing an application to deviate from its normal operating behavior, and attempted to elevate privilege, there would be no direct method for doing so. As currently implemented, SourceT may allow an attacker under very specific conditions to read and write to portions of memory controlled by code running at higher privilege levels, resulting in the potential for altering the expected behavior of that code. However, such attacks would be limited to utilizing existing code present in the system, and would not allow for the introduction of additional code in to the system.

Should attackers somehow manage to overcome these first two protections and attempt to install a rootkit by altering system binaries or otherwise subverting the boot process, the attempt would fail. Ignoring the sheer difficulty of accessing methods allowing access to the disk or boot loader code, any alterations to system images will be caught by the bootloader at signature verification time. As this verification takes place via a TPM, it is not possible to avoid checking the signature, and the use of the TPM is required to gain access to the key required to decrypt the system binary. Recovery of this value may be possible on a per-system basis, but would require debug access to perform. This level of access is not available on production systems.

As such, the traditional methods used by malware to gain system access and obtain the privileges necessary to install themselves for continued infection do not appear viable.

2. Methodology

Matasano staff spent September 25th, and 26th, 2007 onsite at the Secure64 offices, getting an overview of the key aspects of the SourceT operating system. Each aspect with security relevance was covered, including topics not directly relevant to the goals of this paper. This was done in order to achieve an understanding of the processes that went into the design of the SourceT OS. Knowledge of the larger picture architecture, and not just aspects that were to be analyzed helped identify both flaws and mitigation strategies that might not otherwise have been obvious.

Following onsite meetings, code injection, privilege escalation, and the trusted boot process were all examined in isolation, to determine what weaknesses, if any, were present. Traditional operating system weaknesses and attack vectors were examined, and the defense techniques employed by SourceT to combat them were studied for completeness. Where current implementation differed from the specified architecture, both were examined, and any flaws present in current implementation that were not the result of architectural design were noted.

Where non-architectural flaws were present, these were also noted. Where programming flaws have and have not been architected out of the SourceT operating system, this was also noted.

3. Overview of Secure64 Technology

The SourceT operating system was designed from the ground up to be secure. By taking advantage of the security features contained in the Itanium processor family, Secure64 architected a system that eliminates many of the flaws that plague modern operating systems. Where general purpose operating systems have to provide generic services to meet a wide variety of needs, SourceT provides a limited set of services and interfaces, designed for the specific needs of the applications being deployed on it.

General purpose operating systems are designed to work against a lowest common denominator. If a system can't guarantee the presence of a TPM, specific MMU design, or a trusted boot process, it is often the case that these advanced features go unused. By targeting a specific processor architecture, with a known set of capabilities, SourceT takes advantage of modern advances in the security space, with none of the baggage that other operating systems have to contend with.

4. Areas of Analysis

Matasano evaluated the SourceT architecture against three areas of security vulnerability: code injection, privilege level escalation and alteration or subversion of the trusted boot process. In many cases, the architecture specifies multiple layers of protection in order to achieve its security goals. These layers were examined both in isolation, and in conjunction with each other.

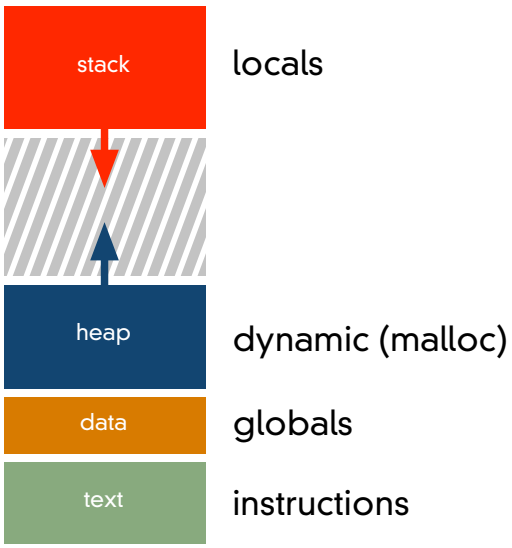
4.1 Code Injection

The ability for an attacker to inject or otherwise manipulate the runtime of an application on the system is the cornerstone of traditional attacks against operating systems. By taking advantage of application structure or programming flaws, attacks inject executable code into networked systems. The ability to execute code on a compromised system opens enormous possibilities for an attacker. They exploit compromised systems to introduce spam relays, spy on system users, or conduct further attacks to elevate their privilege, gaining additional access and further reach in the kinds of activities they can conduct.

4.1.1 Traditional Application Model Code Injection

4.1.1.1 Traditional Application Layout

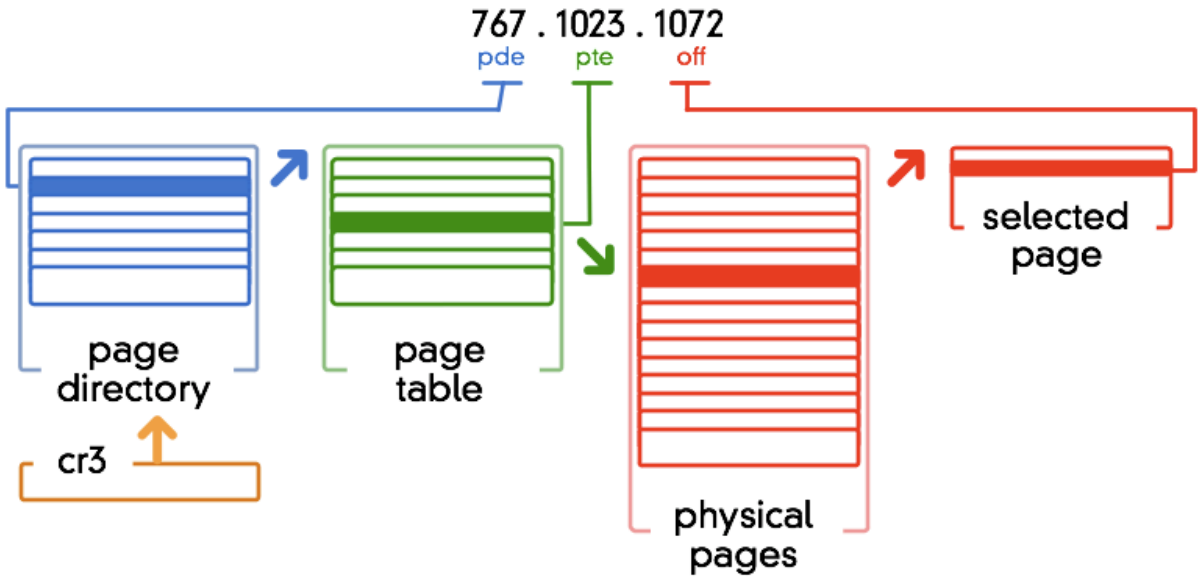
The traditional application model breaks the accessible address space into a number of discrete elements, a number of which can be manipulated by an attacker with the goal of subverting a system.



The stack, used for keeping information related to the functions currently being executed by an application, can be attacked through a variety of methods. Attackers may attempt to exploit certain types of vulnerabilities to alter variables contained in the stack, or alter the call chain itself, causing the execution of arbitrary code upon a return from a function.

The heap contains dynamically allocated memory, as well as any libraries that may be loaded. Attackers may attempt to alter variables, overwrite function pointers, or alter the headers of the in-band management structures in order to execute arbitrary code.

The IA32 architecture provides a straightforward two-tier memory management model:



X86 addresses encode a page directory index and a page table index, allowing for 32 bit address mapping in a sparse address space. Operating system processes use the CR3 register to store the location of the page directory, the root of address lookups for each process.

Memory protection under IA32 is tightly coupled with virtual memory address mapping. The security capabilities provided by the IA32 MMU are not regarded as particularly flexible. Only recently have mainstream operating systems taken advantage of page protection bits, such as Win32 DEP or OpenBSD W^X.

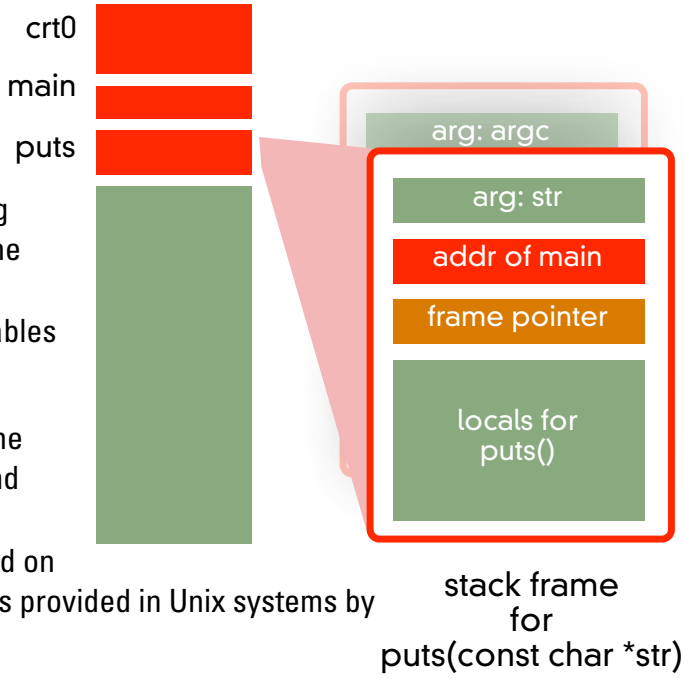
Traditionally, both the stack and the heap are writable, readable and executable.

4.1.1.2 Traditional Stack Architecture

4.1.1.2.1 Design

The design and implementation of a call stack architecture has a major impact on the ability of an attacker to take advantage of certain classes of exploits, such as *buffer overflows* and *format string attacks.* Many modern operating systems have attempted to address the shortcomings of the traditional application stack design, with varying degrees of success.

In a traditional application model, a *stack frame* is generated for each function call.



Stack frames are pushed on to the *call stack*. The call stack contains any information relevant to the function being called. This includes the arguments to the function, the address to return to when execution completes, and any local variables in the function itself.

The stack is sized at run time based on the number and size of the local variables and any arguments passed to the function. Additional variables can also be allocated on the stack dynamically; this functionality is provided in Unix systems by the alloca() call.

A call by a function to another function causes an additional frame to be generated and pushed on the stack. When a function has completed execution, register and control state for the previous function is restored, and the application continues execution at the address specified on the stack.

4.1.1.2.2 Flaws

Buffer overflows occur due to programmer error. Variables on the stack are sized statically, and are set by the programmer. An overflow occurs when an operation stores too much data in a buffer. This results in the writing of data into the adjacent buffer. This often causes abnormal behavior in an application, including application crashes.

The typical exploit of a stack overflow takes advantage of the fact that the local variables for a function are stored at addresses lower than that of the return address. If a large enough write can be performed to a variable on the stack, it is possible to overwrite the return address. If the attacker can set the address to point to code of their choosing, it may be possible to execute arbitrary code. The location where code is written is usually the variable being overflowed.

4.1.1.2.3 Countermeasures

Modern stack implementations have attempted to mitigate this problem by marking the page permissions of the stack to be non-executable. This prevents utilizing the stack variable as the container for the executable code. It does not, however, prevent the stack overflow from taking place.

If code can be located in an alternate location where the page permissions allow for execution, it may still be possible to execute arbitrary code. Other methods of attack have taken advantage of library calls that can perform the tasks required by the attacker. These are generically referred to

as *return to libc* attacks, as the return address written on the stack points to a function within a library. As libc is typically mapped in to the address space, the method gained the name of the most commonly used code vector. *Address Space Layout Randomization* is one technique employed to counter this attack, although research has suggested it is ineffective in 32-bit machines.[1]

A method employed to prevent the altering of the stack return address is the use of stack *cookies* or *canaries.* By placing random values in the stack, which are verified upon return, it is possible to detect when the return address has been altered.

Other techniques have been employed to relocate potentially exposed variable types location on the stack, to limit their ability to overwrite adjacent variables. [2]

Program flaws can also allow the targeted overwriting of variables on the stack. *Format string exploits* take advantage of an erroneous programming pattern often employed. Vararg functions, like *printf()* ,are designed to take a variable number of arguments: typically a format string, and a series of variables. The format string specifies the print format of the corresponding variable being passed. Vararg functions work by reading unnamed variables directly from the stack, using a simple interface. It is not possible to determine the number of arguments being passed when varargs are being used. If a programmer allows an attacker to supply a format string, they can specify more format modifiers than are being passed by the function. This can allow the attacker to read or write values contained on the stack.

Flaws of this type are typically identified at compilation time.

---

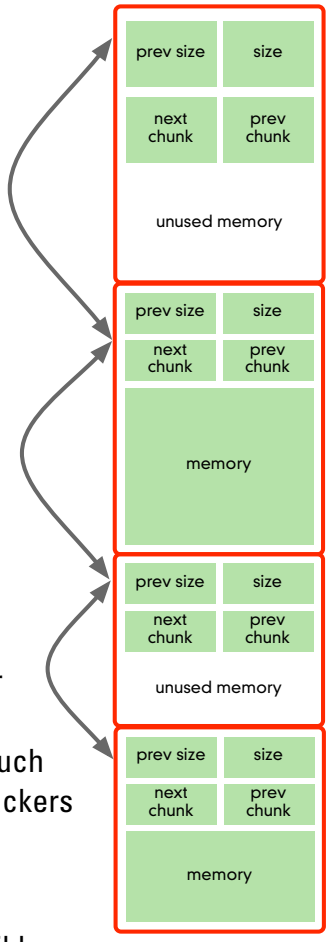
1 http://www.stanford.edu/~blp/papers/asrandom.pdf

2 http://www.research.ibm.com/trl/projects/security/ssp/

4.1.1.3 Traditional Heap Architecture

4.1.1.3.1 Design

The heap is the portion of memory utilized to map dynamic memory allocations.



Traditional heap implementations utilize in-band bookkeeping mechanisms to manage allocations and deallocations. The heap tracks metadata to allows it to allocate new regions, free blocks for future reallocation, and coalesce adjacent blocks upon freeing. Coalescing is performed by adjusting pointer values in chained blocks.

The time and space performance characteristics of dynamic allocators are a rich area of computer science research. Program heap implementations in mainstream operating systems are not designed for resilience against memory corruption; they are designed to make maximum use of available memory, with minimal "fragmentation", at the lowest runtime cost possible.

4.1.1.3.2 Flaws

When copy or write operations for a heap-allocated buffer exceed its allocated size, buffer overflows similar to those in the stack occur. After running past the buffer's own memory, other allocations can be corrupted. Attackers exploit this condition to alter sensitive variables, such as function pointers, to hold attacker-controlled values. This leaves attackers with control of execution flow.

Due to its in-band management structure, attacker can often overwrite metadata for adjacent allocations. When done carefully, it may be possible for an attacker to write arbitrary values to arbitrary locations upon a free() of a block.

Typical programming problems exist around the tracking of pointers to dynamically allocated memory segments. A common error that occurs when two locations hold a pointer to the same memory is that one (A) frees, while the other (B) still uses it. Location (B) is now writing to a stale allocation, which will be reallocated and populated with different data. Attackers exploit these situations to gain control of program data. Another common error is where two different code paths attempt to free a single allocation, resulting in a *double free* error. With double-frees, an attacker again may have the opportunity to control the heap management header, allowing arbitrary pointer writes.

As pointers are reused following a free, there may also be situations where a stale pointer could be utilized to populate a legitimately allocated block, which when the contents are dereferenced or freed, could result in code execution.

4.1.1.3.3 Countermeasures

A common mechanism utilized to limit the damage of heap overflows is to use the NX bit to disable the execution of code on pages mapped in to the heap. This prevents an attacker from executing code they've introduced directly in to the heap. This does little to limit the impact of overwriting function pointers to access functions already mapped to executable pages, however.

Attempts at addressing some of the impact of heap overflows have been undertaken in main stream operating systems, such as Windows DEP. These solutions have, however, been bypassed by attackers under certain conditions.[3]

4.1.1.4 Traditional Memory Initialization and Utilization

In native code, variables don't take on values until program code explicitly assigns them. Because the C programming language doesn't force developers to provide variables with initial values, a common source of C implementation flaws occurs.

4.1.1.4.1 Design

At application initialization time, the memory area containing the stack is in a generally unknown state. When functions are called, the values corresponding to the addresses of local variables may contain arbitrary values.

In the heap, when a block is first allocated, it contains whatever value the kernel allocator has initialized it to. This is often all null bytes. In debug modes, compilers often initialize this to some sort of test-pattern, which allows an application developer to know when a bad pointer causes a dereference into an uninitialized area of memory.

Reused blocks are not reinitialized by all allocator implementations prior to returning a pointer as the result of an allocation. Often, the allocation of a buffer of similar size to a recently freed one will result in the return of the same piece of heap memory. The contents of this memory will be similar to that of the buffer prior to its being freed.

4.1.1.4.2 Flaws

As the result of programmer error, failure to initialize each variable being used can lead to vulnerabilities. If an attacker can determine a way to populate stack or heap contents with known values, and can trigger a condition where a subsequent function call will use or dereference the contents of a variable without initializing it first, the attacker can overwrite arbitrary memory locations, or otherwise affect program logic.

4.1.1.4.3 Countermeasures

Flaws of this nature tend to be the result of developer error, and are application specific. Generic methods of exploitation have not yet been uncovered; the class of uninitialized variable

---

[3] Alexander Anisimov

"Defeating Windows XP SP2 Heap protection and DEP bypass", http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf

vulnerabilities is fairly new. Compilers can provide warnings in many cases where a variable is uninitialized, but perform poorly in inter-procedural situations, where pointers are passed to another function, and due to exceptions, are not initialized prior to return.

4.1.2 Secure64 SourceT Application Code Injection

4.1.2.1 Summary of analysis and findings

The SourceT stack is robust against traditional style stack overflows, which seek to overwrite the return address contained on the stack. As SourceT uses the RSE to store this information, backed by memory at a higher privilege level inaccessible to any application, there is no mechanism which allows an attacker to write to these values via overflow.

The heap is similarly robust against traditional heap overflows. These typically take advantage of the placement of book keeping data within the heap itself. As SourceT places its control information outside of the heap, in an area inaccessible via overflow, it is not possible to alter these values maliciously.

Neither the heap nor the stack contains executable pages. As such, attacker specified code cannot be introduced in to the system. The use of a split instruction and data TLB by the Itanium architecture, and the lack of any mechanism in the SourceT architecture to create entries dynamically in the instruction TLB makes the introduction of code a complex process that an attacker would be unable to perform without being able to introduce code in to the system.

While it may be possible for an attacker to alter pointer or other variables in the stack or heap, or take advantage of uninitialized memory flaws in order to alter application flow, the likelihood of performing a successful attack within the functionality already present in the application is low.

Barring implementation flaws, the likelihood of an attacker being able to inject code is extremely low. As architecturally specified, currently known means of code introduction and execution would not be successful.

4.1.2.2 Overview of Itanium capabilities used by SourceT

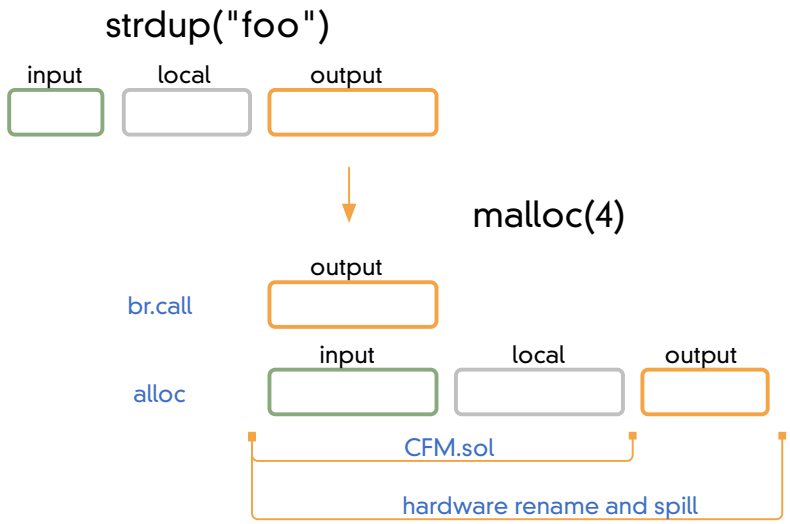
The SourceT architecture makes use of a number of security relevant features found in the Itanium processor. These features provide the building blocks for their architectural solutions to traditional code injection problems.

4.1.2.2.1 The Itanium Register Stack Engine

Itanium provides a hardware-managed register stack mechanism called the Register Save Engine (RSE). The RSE allocates and renames up to 96 “stacked” general purpose registers so that each subroutine within a program receives its own set of visible registers. The lowest visible registers contain arguments passed to the function. The highest visible registers are used by the function to pass arguments to those functions it calls. The remaining registers are accessible only within the function. The use of a hardware managed register eliminates the need for function callees to

spill and restore register states [4], simplifying register management, enhancing performance and, by side-effect, reducing the likelihood of certain classes of attacks.

As outlined above, each subroutine has three subroutine-specific groups of registers: "inputs", which store the arguments provided by a subroutine's caller, "locals", for local variable storage, and "outputs", which provide arguments to further called subroutines. In the Itanium calling convention, the `alloc` instruction portions and renames registers from the dynamic register file.



The RSE provides explicit hardware spill and fill. When the dynamic register file is exhausted, the RSE spills GPRs to a region of memory. This memory region can be at a higher privilege level that than the function itself.

As a result, it is possible to utilize Itanium's RSE to create a subroutine activation state management scheme that is out-of-band. The use of out-of-band state management has been shown to be an effective defense against stack overflows. [5]

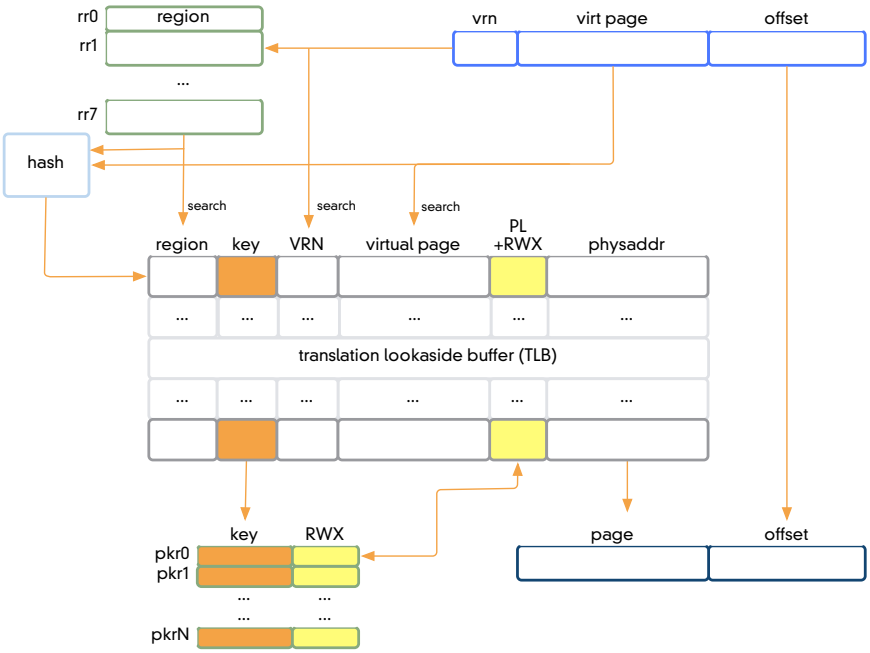
4.1.2.2.2 Protection Keys and the Itanium MMU

The Itanium MMU and address translation model differs substantially from the classic IA32 model:

---

[4] Intel Itanium Architecture Software Developer's Manual Volume 1, Revision 2.2: Application Architecture. Pg 1:16.

5 The DynamoRIO Collaboration. http://www.cag.lcs.mit.edu/dynamorio/



In Itanium, address space is divided into "regions", shareable among processes. Regions are subdivided into variable width pages. Region and page identifiers are concatenated to form the 85 bit virtual address presented to the Itanium MMU.

Itanium also supports the concept of protection keys. The system associates a set of active protection keys in the hardware protection key registers (PKRs) with each task, and each page is tagged with a 24-bit protection key. When the associated protection key is not present in a PKR, any access raises a fault; otherwise, bits associated with the protection key can independently remove read, write, and/or execute permission from the page. This is a flexible primitive for building enhanced OS-level memory protection features, with hardware enforcement. This can be used, for example, to allow each task on an Itanium system to maintain separate access rights for shareable memory pages.

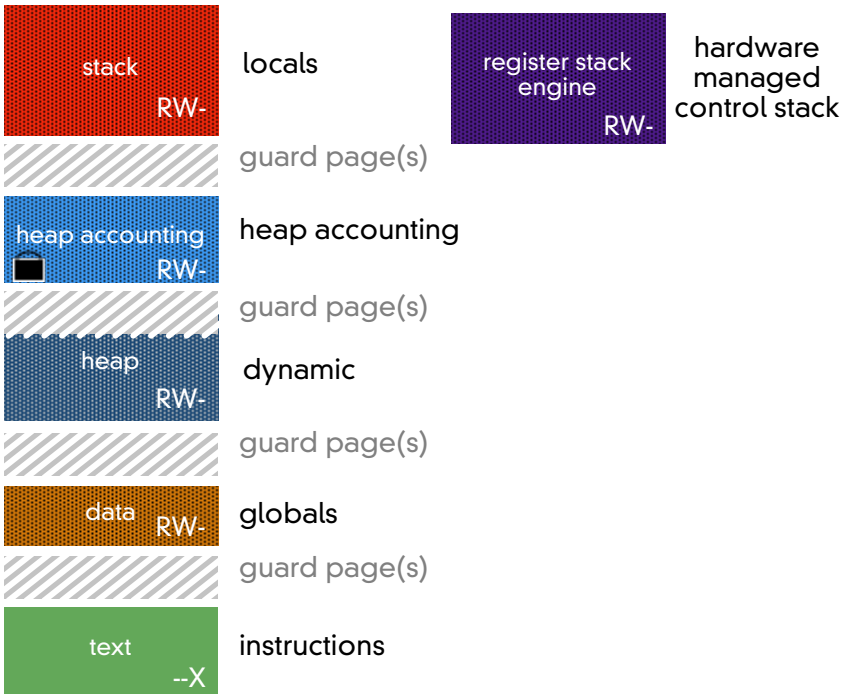
Protection keys are a useful feature of Itanium, but are not heavily exploited by mainstream operating systems. A key feature of the Secure64 SourceT operating system is the use of protection keys to guard runtime state.

Additional information about the design and implementation of the Itanium memory management unit is available in the Intel Itanium Architecture Software Developer's Manual, Volume 2.[6]

4.1.2.3 SourceT Application layout

The Secure64 SourceT operating system application model is similar to a traditional one, with a number of changes made to eliminate common implementation flaws.

[6] http://www.intel.com/design/itanium/manuals/245318.htm



The memory based stack in the SourceT operating system only contains local variables related to function execution. A hardware RSE contains any function arguments, as well as the information required to return control to the previous function. While previous memory stack frames are accessible to the current function, the contents of the previous RSE entry are not, because they are stored at PL0. Use of the Itanium compilers results in execution control information being stored out-of-band, and as such is not subject to accidental overruns or even targeted "write-8-bytes" vulnerabilities, such as the QT4J Quicktime vulnerability.

The heap in the SourceT operating system uses an out-of-band memory management system. Access to the memory management structures are restricted using Itanium protection keys. Program heaps are thus protected using fine-grained, page-specific protections managed by the hardware and the operating system.

Ultimately, the use of protection keys to protect *compartments* will be available as a general purpose facility to applications. Developers will be able to protect critical pieces of data to prevent tampering, even in the event of an otherwise-exploitable implementation flaw.

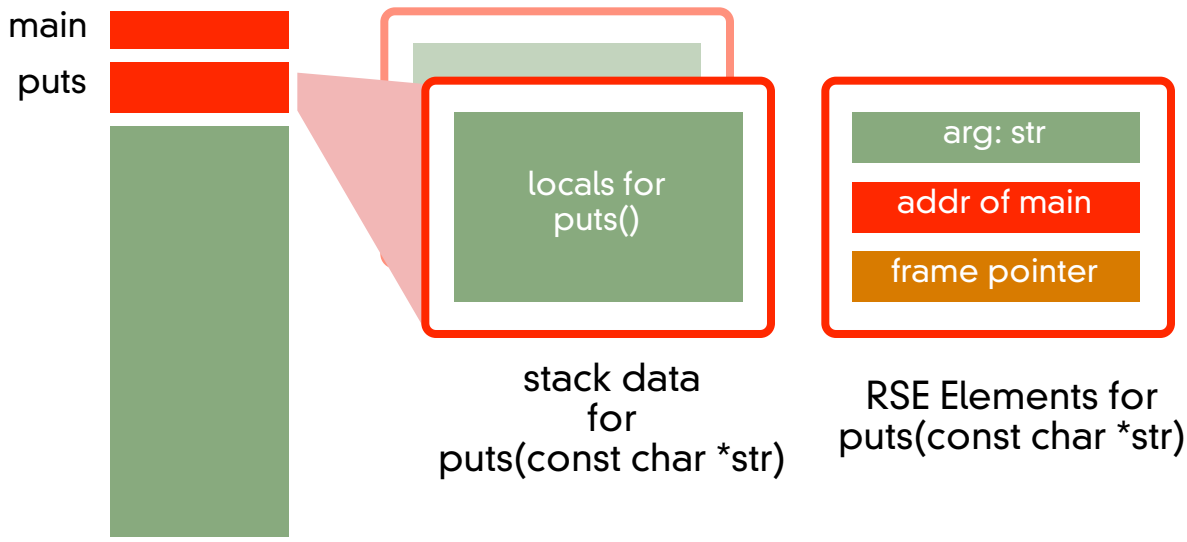
The standard memory model for SourceT programs is hardened. Each application section is separated by unmapped guard pages. As with OpenBSD W^X, neither the heap nor the stack areas are executable. Instruction pages are only mapped in to the instruction TLB, and as such can not be read or written by the application.

4.1.2.4 SourceT Stack Architecture

The SourceT stack architecture takes advantage of the RSE mechanism provided by the Itanium, effectively separating control information and function arguments from the local variables stored on the stack.

4.1.2.4.1 Stack Design

The stack architecture employed in SourceT differs from that of a traditional architecture:



Under SourceT, stack control information is passed in the Itanium RSE. The RSE stores the function arguments, and any control information, in a set of registers. [7] Function local variables are still stored in a traditional, memory based stack. The stack itself is non-executable.

As control information is stored in registers, instead of being stored in memory adjacent to local variables, it cannot be written to in the event of a buffer overflow. Under SourceT, RSE operations occur at an RSE specific privilege level, so even in the event that the RSE contents had been spilled to memory, they remain inaccessible at other processor privilege levels. In SourceT, unprivileged RSE control instructions are also forbidden, and are detected by scanning application code.

Current versions of the Itanium compiler do not support any of the extended stack protection mechanisms available on many other GCC supported platforms, including stack cookies and variable reordering. In the SourceT stack layout, using cookies to protect the control information on the stack is unnecessary, as it is not present in the stack.

4.1.2.4.2 Potential Flaws

The SourceT stack implementation does not protect against the overflow of one variable buffer into another variable buffer on the stack. Should the stack contain variables that affect program logic, such as loop counters, or flag values used to control execution of code within the function, it remains possible for an attacker to alter program logic. If function pointers are present on the stack, it remains possible for an attacker to cause calls to alternate functions.

---

7 Intel Itanium Architecture Software Developer's Manual Volume 1, Revision 2.2: Application Architecture. Pg 1:43.

The SourceT stack implementation also does not eliminate or guard against varargs format string attacks. If situations exist where a user can supply a format string to a printf()-style function, it may be possible to read the contents of variables on the stack. As SourceT does not support the *%n* modifier, however, it is not possible to write values back on to the stack.

If pre-existing system or library calls exist that contain simple function interfaces, it might be possible for an attacker to call them by overwriting function pointers in the heap or stack. As privileged calls are extremely limited, and the application model doesn't provide for multiple processes, there may not be an existing surface where an attack of this nature would result in any meaningful injection outcome. Flaws of this sort can only exist as the result of programmer error, and even then, only in very specific application designs.

4.1.2.4.3 SourceT Stack Code Injection Summary

As control variables are contained within the RSE, traditional stack overflows can not succeed.

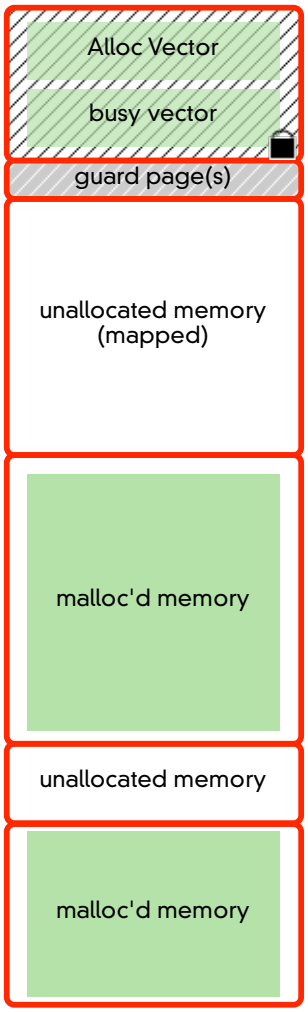
While it may be possible for an attacker to alter application flow in order to induce the system to perform operations other than those it intended, it does not appear possible for an attacker to directly inject executable code into the stack. No stack pages are marked executable.

Altering the permissions of pages to be executable would require the execution of PL0 privileged instructions. Once marked executable, further privileged instructions would be required to move a given page into the instruction TLB before it could be executed. There are no accessible system calls that would allow an application to do this. Code injection methods to achieve this present a chicken-and-egg problem, in that injecting code would require introducing code into the system.

4.1.2.5 SourceT Heap Architecture

4.1.2.5.1 Heap Design

In the current implementation of SourceT, all non-allocated memory is mapped into the heap. SourceT does not support paging, so there is a one-to-one mapping between virtually addressed pages and physical pages. A given virtual address always points to the same physical address.



The heap management currently implemented by the SourceT operating system utilizes an out-of-band bookkeeping mechanism. A separate page, isolated from the heap itself, contains two vectors that are used to track allocated pages. Writes to these vectors are protected using Itanium protection keys.

The protection keys controlling access to heap metadata are loaded and unloaded by the heap allocator itself. The result is a "least-privilege" system, where SourceT programs revoke their own access to the heap until they enter the allocator code path. Absent bugs in the allocator itself, SourceT programs cannot directly attack allocator state.

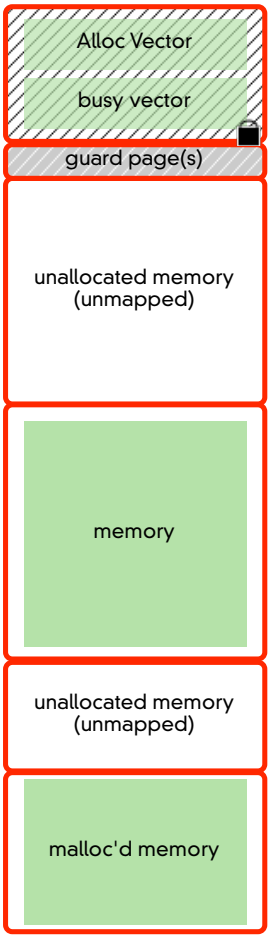
The SourceT allocator explicitly considers double-frees. Double frees are silently ignored when a block that has not been reallocated is freed for a second time. However, if a block is reallocated, SourceT can no longer differentiate between valid and invalid frees.

Unallocated portions of memory are still mapped into the virtual address space, and can be directly addressed by an application regardless of whether or not they've been allocated.

The future, architecturally specified heap implementation for SourceT will alter two areas. First, the entirety of available physical memory will not automatically be mapped in to the heap. Only upon allocation will an entry be placed in the TLB to allow access to physical memory via a virtual memory translation. Attempts to write to unallocated blocks will result in a fault.

The second area of change involves eliminating the reuse of pointer values. While traditional allocators will return the same pointer repeatedly if the block is available, Secure64 will be implementing a system where each allocation returns a unique pointer. This will eliminate situations where a stale pointer can address a currently allocated block, as the translation between a free virtual and physical address will be invalidated.

Further protection mechanisms will be put into place to prevent tampering with book keeping records. Currently, book keeping is only accessible to the malloc routines, which protects the pages to prevent a stray pointer from inadvertently overwriting information. Ultimately, allocation will take place at a higher privilege level, and will require a protection key inaccessible to PL3 processes.



4.1.2.5.2 Potential Flaws

If pre-existing system or library calls exist that contain simple function interfaces, it might be possible for an attacker to call them by overwriting function pointers in the heap or stack. As privileged calls are extremely limited, and the application model doesn't provide for multiple processes, there may not be an existing surface where an attack of this nature would result in any meaningful injection outcome.

While simple double free attacks have been eliminated, the current implementation of SourceT's allocator does have stale pointer problems that could allow a stale pointer to manipulate a block currently being utilized, potentially altering variables. In the eventual implementation, as pointer values will not be reused, this should no longer be an issue; stale pointers will no longer point to valid regions of memory once freed, and will cause an exception when dereferenced.

4.1.2.5.3 Heap Overflow Summary

As is the case with data on the stack, it does not appear possible for an attacker to directly inject executable code into the heap. No heap pages are mapped as executable, and doing so would require the introduction of code in to the system. While data flow may be altered, and it may be possible to cause application specific flaws to arise, there does not appear to be a method of introducing attacker specified code in to the heap.

4.1.2.6 SourceT Memory Initialization

4.1.2.6.1 Design

As currently implemented, SourceT does not have any architectural elements to address these programming errors. Compiler errors are generated for any class of uninitialized variables it is capable of detecting. Heap and stack contents are not reinitialized when the are reused.

Future versions of SourceT intend to address the stack and heap initialization issues. For example, the next major release will initialize heap memory to a known pattern when it is allocated.

4.1.2.6.2 Memory Initialization Flaws + Summary

Memory initialization errors can still exist in the SourceT architecture. This could allow for writes to arbitrary points in application space, if an attacker has the ability to influence stack or heap contents, and the knowledge of how to induce uninitialized memory accesses. This requires situations where the attacker can influence memory contents in ways that will allow them to control a specific value that will be accessed. Such attacks are extremely difficult to identify, and exploiting them is highly application specific.

4.1.2.7 Viability of Code Injection in the SourceT Architecture

Traditional methods of code injection via the stack or heap are not viable for a number of reasons. The lack of control information being stored in-band on the stack or the heap eliminates overflows which attack these values.

The lack of executable pages in the SourceT operating system further complicates code injection. Even if an attacker were to alter a function pointer or cause an alteration to code execution, no pages are marked executable that can be written by the attacker. Further restricting the ability to introduce code that can be executed is the use of split instruction and data TLB's by the Itanium architecture.

The lack of mechanism for an attacker to introduce arbitrary code in to the SourceT operating system presents an extremely high bar over which an attacker would need to pass in order to even begin to attempt to escalate privilege in the operating system.

4.2 Privilege escalation

An attacker, upon gaining access to a system, will often attempt to elevate the privilege level they have on the system. This is done for a variety of reasons. Elevated privilege levels allow the attacker to perform activities that normal users might not be allowed, such as accessing the files of other users, listening in promiscuous mode on network interfaces, or installing drivers or kernel extensions. Often, elevated privileges are required to cover the evidence of an attack, by altering logs or other auditing facilities. Finally, an attacker may need elevated privileges in order to install a backdoor mechanism to ensure they have access that can survive across a restart.

Attackers will normally look for flaws in system calls, drivers, or other services which execute at elevated privileges, and perform attacks similar to those performed against normal applications, including code injection attacks.

4.2.1 Traditional model privilege escalation

In traditional operating systems, attackers looking to escalate privilege levels will look to services which run at elevated privileges, flawed system calls which can be attacked via overflows or pointer manipulation flaws, or look to poorly designed drivers or other privileged code. As system calls and drivers all execute at PL0, flaws in these result in an attacker having absolute control of the operating system.

4.2.1.1 System Calls

4.2.1.1.1 Design

On most operating systems, system calls are implemented by having library stubs within standard C libraries. These stub take parameters (e.g. the name of the file), and store them in the appropriate places (typically registers), along with the appropriate system call number and then generate a kernel trap or interrupt.

When this specific interrupt is called, the kernel will look up the system call to be executed (via the system call number), and then call it, passing in the arguments passed in via the stub.

System calls are executed at supervisor privilege. As such, any flaw in a system call that can be used to execute arbitrary code does so at a maximum privilege level on the system.

4.2.1.1.2 Flaws

As system calls run in kernel mode, but take user supplied data from userland processes, they are sizable part of a kernel's attack surface. Attacks that can impact system calls include memory trespass vulnerabilities like buffer overflows and integer overflows; arbitrary memory reads and writes via unchecked pointer values; and other forms of attack. In recent years this has become an increasingly common way in which attackers escalate privileges after they obtain interactive access as a lesser privileged user.

4.2.1.1.3 Countermeasures

The use of system calls determined to be privileged are typically restricted to a certain class of user or application roles in a system.

Some operating systems further restrict the ability for users to utilize system calls via role based access control mechanisms, which may restrict an applications ability to perform

4.2.1.2 Privileged Code and Services

4.2.1.2.1 Design

In traditional operating systems, code is executed either in user mode, or in kernel space. Within user mode, privileged users may exist that are allowed to perform restricted activities. This may include configuring portions of the operating system, writing to physical devices via drivers, or loading system extensions. There are few limitations in the activities a privileged, or "superuser" can perform.

Many systems allow individual applications to run with the privilege of this superuser. This may allow them to access the files of other users, or perform other privileged operations programmatically. Drivers, which communicate with physical devices on the system bus, are typically designed as in-kernel services, running at supervisor privileges.

4.2.2 Secure64 SourceT Privilege Escalation

4.2.2.1 Summary of analysis + findings

SourceT system calls primarily execute at a reduced privilege level. By reducing the number of lines of code executed at higher than PL1 privilege, the potential for flaws is reduced, and the audibility of the code increases.

Further verification of trusted call paths restricts the ability of an attacker to even call privileged functions. Even in the event of a programming error, an attacker should be unable to perform the call, thus severely restricting the likelihood of attack.

At the time of analysis, a mechanism was not in place to ensure that invalid pointers could not be passed to system calls. If an attacker could induce a system call to be executed with a pointer to a portion of memory only accessible to PL1 code, it may be possible to read from that location, or write arbitrary data to that location. Secure64 intends to add verification of such pointers.

4.2.2.2 System calls

4.2.2.2.1 Design

All SourceT *service calls* use a call gate implemented by stub Itanium code bundles, executed at PL0. The stub loads the service call number, determines the privilege level, and calls out to the service call being executed. Privileged service calls are only allowed for verified *trusted paths*.

Return privilege levels are enforced. A service call cannot return control to an application at a higher privilege level than the one they entered the service call at.

A majority of the service call code executes at PL1. In the architectural specification, calls to PL0 service calls are only allowed for trusted callers. The caller is checked for identity, whether or not the requested call is authorized for that caller, and whether the expected sequence of activities prior to the call have taken place. Any abnormality results in the service call being denied.

4.2.2.2.2 Potential Flaws

At the time of analysis, a mechanism was not in place to ensure that invalid pointers could not be passed to system calls. If an attacker could induce a system call to be executed with a pointer to a portion of memory only accessible to PL1 code, it may be possible to read from that location, or write arbitrary data to that location.

Performing this type of attack would require the ability to execute arbitrary code, or in some way manipulate an application to perform the call using attacker supplied pointer values. Doing so would depend on the presence of programmer error. Performing an attack of this nature remotely would be difficult. It would not result in the execution of arbitrary code at PL1, but may allow for the overwriting of function pointers or other flow related control information that could alter the expected behavior of code executing at PL1.

A simple example to this might take advantage of the s64_readConnection() or s64_writeConnection() IO functions. By passing pointers in the buffer arguments, an attacker may be able to write to pieces of memory, using attacker supplied data, or cause the transmission of PL1 accessible memory over the network. This would be limited to non-executable memory that also is not protected by a protection key.

Remediation of this problem should be fairly straightforward. Callee functions which take pointer arguments must validate that the pointer being passed points to the address space of the caller. If it does not, the call should fail.

4.2.2.2.3 Summary

While the reading and writing of arbitrary memory locations only accessible to higher privilege levels is a serious flaw, it does not appear to present an opportunity to execute code at this higher privilege level. It is still not possible to have privileged code transfer execution to code introduced by an attacker, as outlined in the code injection analysis. The execution surface available to the attacker is limited to that of code already present and accessible at the higher privilege level. Making meaningful calls to these interfaces, with arguments that are proper and allow for the dynamic creation of some sort of maintained privileged access would be non-trivial.

4.2.2.3 SourceT privileged code and services

4.2.2.3.1 Design

In the SourceT operating system, a minimal portion of code exists that runs at the highest privilege levels (PL0). Currently comprising approximately 4,000 lines of code, access to it is extremely limited.

A majority of driver code executes at PL1 privilege, and operates on virtual addresses. Specific driver activities which require access to physical addressing call support bundles which execute at PL0.

All applications run at PL3 privilege. There is no concept of a “privileged” application. Service calls are executed at non-PL0 privilege levels.

IO processing occurs on a dedicated processor at PL1. All communications between the PL3 application and the IO processor are asynchronous. Disk access is performed by the IO processor, while the calling process blocks waiting for completion.

4.2.2.3.2 Potential Flaws

An attack surface involving unchecked pointers, similar to that of service calls, may exist for drivers that are accessible via system calls at PL3, or from the IO subsystem running at PL1. The minimal PL0 surface utilized by drivers should reduce the risk that implementation flaws that would allow arbitrary reads and writes of PL0 memory exist.

4.2.2.3.3 Privileged code and service summary

The minimal footprint of driver or other code executing at a privileged level in the SourceT operating system minimizes the likelihood of programming error leading to elevation of privilege.

4.2.2.4 Viability of Privilege Escalation in the SourceT Architecture

While the ability to read and write to arbitrary memory locations only accessible to higher privilege levels may exist, it does not appear to present an opportunity to execute code or elevate the privilege of an application.

The use of multiple privilege levels, in conjunction with a minimal number of lines of code running at PL0 results in a system that can be easily scrutinized for flaws, reducing the likelihood of security related ones being present in production code.

4.3 Boot Process

In the event of a compromise, followed by privilege escalation, attackers often attempt to backdoor a system, in order to ensure future access. This can be as simple as introducing additional applications on to the file system, altering the boot loader, or modifying existing binaries.

Traditional operating systems tend to not have trusted boot systems. Many restrict access to critical binaries by unprivileged users, but for the most part, privileged users can perform activities that alter or modify aspects of the boot process that would allow for persistent access.

4.3.1 Traditional Boot process

4.3.1.1 Design

Traditional operating systems utilize a BIOS, Bootprom or other mechanism to handle the initial aspects of system boot. This is typically a simple bootstrapping service, which performs the basic set of initialization required to transfer the boot process to the operating system bootloader. In most cases, no special checks or verification takes place on the operating system loader. Whatever is present on disk, flash or other boot medium is loaded and executed.

Modern PCs have replaced the simple BIOS mechanism with the *Extensible Firmware Interface*, or *EFI*. EFI was designed to overcome the limitations of the BIOS, while providing an extensible

firmware specification that could grow to meet the needs of future operating systems. However, EFI support is not available on a number of popular operating systems, depending instead on legacy BIOS functionality being implemented as an EFI program.

4.3.1.2 Flaws

As the boot process is unauthenticated, the processor has no way to determine if the BIOS contents have been altered. The BIOS, likewise, as no way to determine whether the bootloader being executed has been tampered with. This makes it possible for an attacker to alter the bootloader itself.

Following along the same lines, it is not possible for the bootloader to ensure that the operating system itself has not been tampered with, nor that it does not contain "unauthorized" drivers or extensions. The need to support a wide variety of hardware and functionality mean that unsigned and untrusted components have a means to load themselves into the operating system. While support for driver and executable signing exists in some operating systems, they do not enforce their presence on every application or driver. This allows an attacker to tamper with items that may be loading by the operating system.

4.3.1.3 Countermeasures

The traditional solution to detecting backdoors, rootkits and other forms of persistent malware has been to utilize anti-virus and other detection software, to identify their presence once installed. These methods are based on a combination of signature checking and heuristic based analysis. The effectiveness of these tools varies.

4.3.2 Secure64 SourceT Boot process

4.3.2.1 Summary of analysis and findings

A lack of end-to-end verification of boot components may allow an attacker with local access to tamper with the trusted boot procedure. Performing such attacks remotely, however, would require complete defeat of code injection and privilege escalation security measures, and as such are complex to the point of being non-viable.

4.3.2.2 Overview of Itanium capabilities used by SourceT

The process of trusted boot begins with the firmware signature verification functionality contained in the Montecito and Montvale Itanium 2 processor. At power on, the chip itself performs a signature verification on the first stage of firmware. This first step ensures that the first piece of software executed by the processor is trusted, and that its contents have not been tampered with.

4.3.2.3 Design

SourceT's architecture specifies a boot process that is verifiable from processor power on, to firmware load, EFI transfer to the SourceT bootloader, and through the signature check and decryption of SourceT's components.

The SourceT architecture specifies a *sealed* boot process. In the intended architecture, each subsequent firmware stage is also checked for a valid signature, prior to executing the next stage. In order to allow lower layers to be able to identify tampering at higher layers, the validation of the signature alters the *process control register (PCR)* of the on-board TPM. These registers are read-only, and cannot be altered other than by the TPM. Should any stage fail to validate correctly, the bootloader will not be able to decrypt the executable code, and the system will fail to proceed.

The hardware provider utilized by Secure64 has not implemented the chain of trust extending from the initial firmware load through EFI, to the bootloader. As such, the SourceT bootloader is unable to verify that no modifications were made to firmware, EFI, or if additional programs were launched by EFI.

The remaining components of the OS are signed and encrypted, and verified by the bootloader.

4.3.2.4 Potential Flaws

Due to the lack of verification following the initial firmware check by the processor, the trusted boot process may be subject to tampering, as any stage following the initial firmware through EFI can be replaced or altered by a user with physical access to the machine. Once the hardware manufacturer includes measurement of the bootloader, as specified in the TCG standards, this vulnerability will no longer be present.

As the bootloader is neither signed nor verified in any manner by EFI, a user with disk access can replace or alter the existing bootloader. The bootloader could be altered to extract the AES keys, which could then be used to decrypt and disassemble the operating system and application code. At this point, an attacker could begin to alter it.

Achieving this without physical access would be extremely difficult. As an attacker is unlikely to be able to inject code and run it at an elevated privilege, it would not be possible to reprogram any stage of firmware flash. While the bootloader and operating system code reside on the filesystem, access to it is also restricted.

By removing the drive containing the loader and system executables, it would be possible to alter the loader to begin to extract keys or otherwise save a decrypted version of the operating system and application, which could then be modified and booted after disabling signature checking and decryption in the bootloader. Performing this attack, while viable, is likely to be extremely difficult.

4.3.2.5 Viability of Trusted Boot Subversion by Remote User

While a user with physical access may be able to tamper with the boot process, performing any of the outlined attacks with remote access would require complete failure of the code injection and privilege escalation prevention mechanisms. Performing an attack of this nature remotely, without the ability to inject code and execute at PL0 privilege levels is of such a high degree of complexity that it becomes a non-issue.

5. Conclusions

At the time of analysis, no architectural flaws that would allow for the injection of foreign code in to the SourceT system were identified. While certain classes of application flaws are still present, they do not provide a mechanism that would allow an attacker the ability to introduce code in to the SourceT runtime. Secure64 has indicated their intention to address these lower severity flaws in subsequent releases of SourceT.

Methods which would directly lead to privilege escalation, or allow a remote attacker to alter the boot process were also not identified.

As such, the traditional methods used by malware to gain system access and obtain the privileges necessary to install themselves for continued infection do not appear viable.

6. Matasano Background

6.1 Corporate Overview

Matasano was formed in 2005 with offices in New York and Chicago. Our team members average more than 10 years experience in software security and software development. Matasano team members wrote the first published i386 stack overflow, co-founded the ISS X-Force research team, invented IDS evasion, developed optical switching platforms, and found the first software vulnerabilities in embedded iSCSI and Fibre Channel storage appliances.

Every member of our team has presented at the Black Hat security conference, with subjects including:

- The Protocol Debugger, which helps security testers evaluate proprietary protocols by setting breakpoints on and single-stepping protocol messages on the wire as if they were software instructions
- Discovery of 12 new attack vectors in agent-based systems management tools, and the results of a year-long research project that broke the majority of the agent-based market (by revenue share)
- Development of Cross-Site Scripting worms and Javascript malware
- FPGA-assisted AES key cracking
- Intel Virtualization extensions, and the development of the first VT-capable "hypervisor rootkit" that exploits those extensions

Matasano has deep technical knowledge in a variety of specialized areas of security:

- **Experience with enterprise level security assessment**: the team we propose for this project has successfully delivered large-scale penetration tests for high-risk, high-sensitivity kernel networking projects. Matasano was part of the team that audited Microsoft Windows Vista.
- **Experience with custom protocols**: Matasano has in-depth experience working with applications that are built on custom or proprietary protocols. Jeremy Rauch's Protocol Debugger (PDB) was written in response to the challenges of assessing protocols with minimal documentation.
- **Experience with embedded and closed platform software and operating systems**: Matasano team members have experience developing and penetration testing embedded devices. We recognize that evaluating embedded devices and operating systems is different than testing applications written and deployed on off-the-shelf operating systems

6.2 Locations

Matasano has locations in New York and Chicago.

New York (Corporate Headquarters)
44 Wall St.
12th Floor
New York, NY 10005-2413

Chicago (Midwest Headquarters)
846 W. Randolph
Chicago, IL 60661-2114