Implicit Buffer Overflow Protection Using Memory Segregation

Brent Roth and Eugene H. Spafford
CERIAS and the Department of Computer Science
Purdue University
West Lafayette, IN, USA
{broth,spaf}@purdue.edu

***Abstract*—Computing systems continue to be plagued by malicious corruption of instructions and data. Buffer overflows, in particular, are often employed to disrupt the control flow of vulnerable processes. Existing methods of protection against these attacks operate by detecting corruption after it has taken place or by ensuring that if corruption has taken place, it cannot be used to hijack a process' control flow. These methods thus still allow the corruption of control data to occur but rather than being subverted, the process may terminate or take some other defined error. Few methods have attempted to prevent the corruption of control data, and those that have only focused on preventing the corruption of the return address.**

We propose the use of multiple memory segments to support multiple stacks, heaps, .bss, and .data sections per process with the goal of segregating control and non-control data. By segregating these different forms of data, we can prevent the corruption of control data by overflow and address manipulation of memory allocated for non-control data. We show that the creation of these additional data segments per process can be implemented through modifications to the compiler.

***Index Terms*—buffer overflow, memory segregation, memory segmentation, memory protection, multiple stacks, multiple heaps**

I. Introduction

A single process' memory contains several forms of data, non-control data as well as multiple variants of control data, which are used for differing aspects of that process' operations. Despite this, typical systems use a memory organization consisting of only a single unified stack, single heap, a .bss section, and .data section per process for storing this data. This data is typically in contiguous memory within one or two memory segments where an error in the processing of access to one form of data can access not only that form but access and corrupt several others forms of data as well. Since the widespread exposure of the overflow attack in the Morris Worm [1] in 1987, this technique has been exploited by various forms of malware and tools as attackers have used write-what-where conditions [2] to launch buffer overflow [3] attacks to exploit errors in the processing of data. By exploiting these conditions, attackers are able to not only corrupt poorly processed non-control data but multiple forms of control data as well, such as return addresses, saved frame pointers, and program-defined pointers. This allows attackers to hijack the control flow of a process by injecting and executing their own code or calling preexisting functions not included in the original procedures of vulnerable processes as with return-to-libc [4] and return-to-GOT [5] attacks. In addition, as these attacks are allowed to corrupt control data, they are able to disrupt the control flow of processes, causing those processes to crash from a segmentation fault or other error related to corrupted control data, thus making buffer overflow attacks an effective method for performing denial-of-service (DoS) attacks. As studies have shown [6], the use of buffer overflow attacks continues to be a major threat as several exploitable overflow vulnerabilities continue to occupy the list of most dangerous software errors.

By placing different forms of data in their own stacks, heaps, .bss sections, and .data sections residing on separate memory segments, it is possible to prevent accesses to one form of data from accessing another form of data — a memory access to one memory segment cannot access the memory of a separate non-overlapping memory segment. This prevents buffer overflows and manipulation of memory items allocated to non-control data from corrupting control data. As a result, this removes the threat of a large class of control flow hijacking and DoS attacks. We propose modifications to the compiler to allocate additional non-overlapping memory segments per process to support the use of multiple stacks, heaps, .bss sections, and .data sections per process that will be used to segregate control and non-control data in these memory segments to accomplish this goal.

II. Related Work and Motivation

We are concerned not only with the use of buffer overflows and memory manipulation to perform control flow hijacking attacks but also in their use to perform DoS attacks. Multiple methods exist that attempt to mitigate the buffer overflow problem, but they primarily focus on control flow hijacking attacks.

Some architectural and operating system approaches [7,8,9] attempt to prevent the execution of code injected during an overflow by designating pages of memory as non-executable. While these have been effective against attacks dependent on injecting code to memory located in non-executable pages, such as the stack, heap, .bss section, and .data section, these methods require the enforcement of specific rules for program layout with regards to separating code and data and are unable to protect memory pages containing *both* code and data. Additionally, these approaches do nothing to prevent the corruption of control data by buffer overflows. This has not only enabled attackers to still circumvent these methods [4,10] and failed to prevent buffer overflows from performing DoS attacks but causes successful mitigations of control flow hijacking attacks to result in the corrupted process crashing from attempts to execute memory on non-executable pages — transforming mitigated control hijacking attacks into DoS attacks.

Other compiler-based methods [16,17,18] are designed to protect control data such as program-defined pointers and the return address by detecting when they have been corrupted so these protections can then terminate the corrupted process, thus preventing the hijacking of the control flow of the process. These methods do not actually protect the data itself but rather protect the process control flow from being hijacked by noting changes to one or more data values. As such, these methods can not only be circumvented [20,21,22], but they do nothing to prevent the corruption of control data, thereby failing to prevent overflows from causing DoS attacks.

Ryan Riley et al [23] proposed an operating system level change to the translation lookaside buffers to create a split memory that separates code and data into different memory spaces. Using this method, even though an attacker can use a buffer overflow to inject code into memory, the system will never use that same memory when fetching instructions for execution. However, similar to the previously mentioned methods and noted by those authors, this method does not prevent the corruption of control data and when successful at stopping an attack it results in the crash of the process, thus leading to a DoS.

Address Space Layout Randomization (ASLR) [15] is another method at the operating system level that combats buffer overflow attacks by randomizing the positions of key areas of the process' memory, such as the stack, the heap, and libraries. By randomizing these positions, buffer overflow attacks attempting to perform control flow hijacking via return-to-libc [4], return-to-GOT [5], and other return-to type attacks are made increasingly difficult to succeed as attackers must guess the locations of all the memory required by their attacks. Often such attacks will guess incorrectly for at least one such position, causing the program to attempt an illegal access of memory that forces the process to crash. However, any control data on the stack remains easily accessible by buffer overflows on the stack as does any control data on the heap by buffer overflows on the heap. Thus, as with the previously mentioned methods, ASLR fails to prevent the corruption of control data as well and as such fails to prevent the use of buffer overflows as methods of performing DoS attacks, and, it, too, transforms successfully mitigated control flow hijacking attacks into DoS attacks.

Another compiler-based method, SSP [19], implements the same class of protections as the previously mentioned compiler-based methods, but adds an extra layer of protection in that it rearranges program-defined pointers on the stack so that they reside at lower addresses in memory than local variable buffers thereby preventing the overflow of those buffers from directly corrupting the program-defined pointers. Unfortunately, this extra layer cannot be used to rearrange control data such as the return address, saved frame pointer, program-defined pointers on the heap or program-defined pointers located inside of objects or structs. As such, SSP must resort to the same tactics as the previously discussed compiler-based methods for protecting these instances of control data; detecting when they have been corrupted and terminating the process. Thus, SSP can be circumvented similarly to the other compiler-based methods [20,22] and fails to prevent the corruption of the previously mentioned control data, making it ineffective in preventing buffer overflow based DoS attacks that target these forms of control data and transforming control-hijacking attacks into DoS attacks if they corrupt these forms of control data.

Some literature exists on methods [26,27] using a theme similar to our own, namely the use of multiple stacks per process. However, both these sets of research only focus on the use of one additional stack and the use of that stack in only protecting the return address from corruption. While these methods illustrate the security gained by segregating this one piece of control data from non-control data, they neglect to discuss means by which other control data such as stack frame pointers and program-defined pointers may be secured despite the abundance of literature illustrating the instances of corrupting these forms of control data by attackers [12,14,24]. Thus, while these methods provide protection against the corruption of the return address, any buffer overflow attacks targeting the saved stack frame pointer, program-defined pointers, or other forms of control data can still successfully perform control flow hijacking and DoS attacks.

It is this pervasive lack of capability amongst existing methods to comprehensively prevent the corruption of control data by buffer overflows and the resulting failure of those methods to prevent the use of buffer overflows for DoS attacks that provides the motivation for our work. Note that our compiler approach is orthogonal to the previously mentioned methods that rely on the detection of the corruption of control data or rendering that corruption non-threatening as our approach takes measures to prevent the actual corruption of control data. Our approach is also distinct from those previously mentioned methods that attempt to prevent corruption in that we provide a comprehensive method to protect all control data on the stack or heap rather than only the return address or only program-defined pointers on the stack.

III. Implementing Segregated Memory

Buffer overflows are able to corrupt control data through the exploitation of the improper processing of non-control data buffers. As modern systems provide processes with a single unified stack, heap, .bss section, and .data section per process for data, they force processes to place control data in memory locations contiguous to those of non-control data and accessible by the same memory accesses intended to access only non-control data. This enables an exploit of an instruction intended to access non-control data to access control data as well, often resulting in the corruption of that control data. For this reason, we believe this problem can be mitigated by segregating control data and non-control data into separate typed sections in their own memory segments that will place them in non-contiguous memory locations. As the number of memory segments, stacks, heaps, .bss sections, and .data sections used by a program's processes can be altered during compilation to be supported by the OS during runtime, we propose this be implemented at the compiler level.

A. *Using Memory Segmentation*

As an example, consider the Intel architecture. It allows systems to support thousands of memory segments available both globally and locally amongst their processes [9]. Despite this, modern operating systems use few such segments, with Linux using only six types of memory segments: kernel code, kernel data, user code, user data, task-state, and default local descriptor table [11]. The use of a memory segment requires its segment descriptor be loaded into one of six available segment registers (which allow segment descriptors to be swapped in and out) and that instructions accessing memory allocated to that segment use the segment selector as part of their logical address — a requirement already imposed under the unified stack, heap, .bss section, and .data section implementation, as memory segmentation is still being used (only to a lesser extent). As *segment descriptors are typically created by compilers, linkers, loaders, or the operating system, but not application programs* [9] and *segment selectors are visible to an application program as part of a pointer variable but the values of selectors are usually assigned or modified by link editors or linking loaders, not application programs* [9], we believe the compiler could be modified to create additional memory segments used by a program's processes in a manner transparent to that program and its process(es.) We could then use these additional segments to support the additional stacks, heaps, .bss, and .data we desire. Each segment's characteristics and permissions are designated in the segment descriptors created by the compiler and thus can be used to further customize each segment for our needs, including growth direction and restricting the bounds on memory accesses to each memory segment preventing overflows from one segment into another. Additionally, as memory segments are resizable, processes are *not* forced to waste memory for the sake of data segregation.

B. *Memory Segment Usage*

As illustrated in Figure 1, the compiler needs only to create and modify code to support five new memory segments in addition to the two original memory segments for data usually found in unified implementations. This provides us with a sufficient number of memory segments to support the four stacks, three heaps, three .bss sections, and three .data sections we desire for our proposed implementation of memory segregation.

The first new memory segment, the *control stack segment*, contains only a stack for pushing and popping control data related to the call stack, such as return addresses, saved frame pointers, saved stack pointers, saved segment selectors, saved values for EFLAGS, and longjmp buffers, along with any other related registers. These values and a process' use of them remain largely the same as these values are still pushed and popped to and from a stack in response to CALL, RET, and longjmp type instructions as well as for interrupt and error handling. The only differences that exist for this stack are the use of multiple stack pointers and stack frame pointers which will need to be pushed and popped to and from this stack in support of the process' use of the multiple stacks.

The second new memory segment, the *pointer stack segment*, contains only a stack containing program-defined pointers. Again, this stack is used in the same manner as the unified stack in traditional systems to push and pop data values to and from as they are needed. However, continuing in the idea of memory segregation, this stack is used only for pushing and popping pointer values. Closely related to this is the third new memory segment, the *pointer data segment* which contains a heap, .bss section, and .data section. These, similar to their counterparts in modern unified implementations, store static, global, and heap allocated data, but, as this is the pointer data segment, it only does so for pointers that are meant to be allocated statically, globally, or in a heap.

The fourth and fifth new memory segments, the *anonymous stack segment* and *anonymous data segment*, operate similarly to the pointer stack segment and pointer data segment respectively. However, the existence of these two memory segments is only an exercise in completeness. As previously mentioned, the allocation of data to the various data segments depends upon parsing by the compiler. In the event that the compiler is unable to decipher from its parse whether a given memory allocation is for call stack, pointer, or non-control data, which will be discussed more in later sections, such allocations are made either to the anonymous stack segment or the anonymous data segment, depending on whether the allocation was to a stack, heap, .bss., or .data section.

With the five new data segments handling their respective segregated forms of data, we are left with the two memory data segments used for data in unified implementations. Given the forms of data handled by the previously mentioned five segments, these two segments are left handling the remaining form of data, the non-control data. As such, we will refer to these two data segments hereafter as the *non-control stack segment* and *non-control data segment*. Similar to the behavior of the newly created stacks and their unified stack counterparts, these memory segments are used to store non-control data intended for the stack, heap, .bss, or .data section respectively with the non-control stack segment handling the former one and the non-control data segment handling the latter three.

C. *Handling Multiple Stacks*

As previously touched upon in the discussion on the control stack segment, each of these stacks we propose requires its own stack pointer and stack frame pointer as the process must be capable of tracking the top of each stack as well as have a reference point within each stack from which to

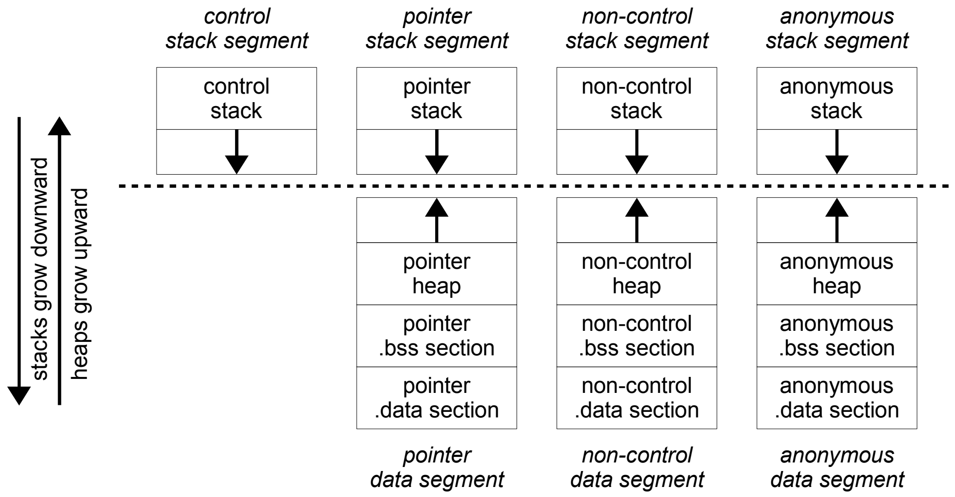

Fig. 1. Segregated Memory Segments

access memory relatively. That is, when pushing, popping, or generally accessing data from a stack, the process must ensure it uses the stack pointer and/or stack frame pointer for that stack. This requirement of four stack pointers and four stack frame pointers largely differs from the requirement of a single stack pointer and stacker frame pointer in modern systems, placing additional demands on hardware, and, as such, warrants additional discussion.

While current Intel architectures only support a single dedicated register for the stack pointer (ESP) and stack frame pointer (EBP), this does not prevent us from using four stack pointers and stack frame pointers per process as processes are allowed to load and store the values of the ESP and EBP registers at will. Thus, the multiple stack pointers and stack frame pointers required by our proposed implementation can be pushed and popped to and from the control stack segment as part of instructions added by the compiler to swap these pointers in and out of the ESP and EBP registers as they are needed. Other general-purpose registers can be used as well to support the use of our multiple stack pointers and stack frame pointers. However, this lack of custom hardware support and additional demands on general-purpose registers does suggest an overhead for this implementation under current Intel architectures. We do, however, believe that, as each stack may not be in use by each function of a process, it may be possible to reduce this overhead by restricting the need to handle these additional stack pointers and stack frame pointers to only those functions of the process that use those particular stacks. That is, the prologue and epilogue of each function would be customized to only handle the stacks used by that function.

D. Handling Multiple Heaps

The handling of multiple heaps is a simpler task than that of handling multiple stacks. Multiple operating systems such as Microsoft Windows [28], IBM's AIX [29], and Linux [30] support memory allocators that use multiple heaps. While these allocators were initially supported to remove the bottleneck of heap memory allocation from multi-threaded programs, their implementations do not restrict the use of multiple heaps to the use of multiple threads. As such, they can be leveraged by the compiler to support the multiple heaps per process required by our implementation whereby the compiler uses their operating system's respective multi-heap allocator by default and replaces instructions in code to mono-heap memory allocator functions with their multi-heap counterparts.

E. Handling Complex Objects (or Structs)

Complex objects (or structs), are by definition, those which contain both pointers and primitive datatypes (non-control data). The use of such complex objects would appear to be counterintuitive, if not impossible, under the proposed segregated memory segments as the object would require the pointers and non-control data be stored in contiguous memory locations within the same memory segment, but that is in direct conflict with the segregated memory segments that require they be stored in separate memory segments. However, as illustrated in Figure 2, there is a simple answer to this predicament. As the goal of the complex object is to retain the relationships amongst its pointers and primitive datatypes, the primitive datatypes can be replaced with pointers referencing primitive datatypes. This retains the relationships between the complex object's pointers and primitive datatypes while ensuring the object is now completely composed of pointers and can thus be allocated to the pointer stack segment or pointer data segment while its primitive datatypes are likewise allocated to the non-control stack segment or non-control data segment.

Unfortunately, this is not the end of the complications. As illustrated in Figure 3, in the case of object inheritance or other forms of overloading, there may exist a complex object that is a child of a non-complex object. This presents a problem as, if we only replaced the primitive datatypes in the memory representation of the complex child object, but do not do the

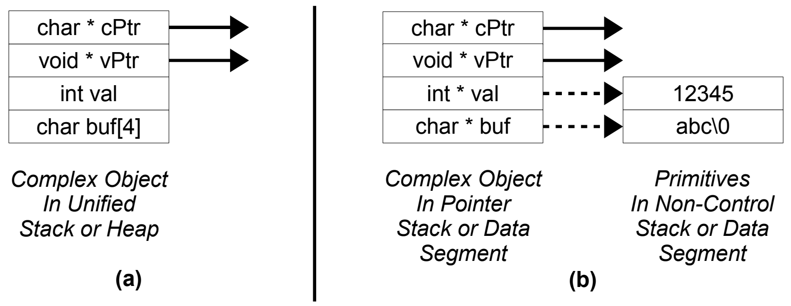
char * cPtr
void * vPtr
int val
char buf[4]
Complex Object In Unified Stack or Heap
(a)
char * cPtr
void * vPtr
int * val
char * buf
12345
abc\0
Complex Object In Pointer Stack or Data Segment
Primitives In Non-Control Stack or Data Segment
(b)

Fig. 2. (a) Complex object pointers and primitives in same memory segment (b) Complex object pointers and primitives in segregated memory segments

```
class NonComplex {
        int val;
        // …
};

class Complex : public NonComplex {
        char * ptr;
        // …
};
```

Fig. 3. Inheritance with complex child object and non-complex parent object in C++

same for the non-complex parent object, then, when given a pointer to a parent object, we may be referencing either a non-complex parent object or a complex child object. Hence, we would be forced to distinguish between the two as indirection is required to access the primitive datatypes of the complex child object but not for the non-complex parent object. The simple solution to handle such cases if for the compiler to replace all primitive datatypes of non-complex objects that are inherited by complex objects in the same manner in which those complex objects' primitive datatypes are replaced. This allows indirection to be used whether in reference to the values of the complex object or its non-complex parent object without the need to know which is being accessed. While this provides some additional complexity, requiring identification of complex objects and tracing of their inheritances, it handles the issue during compilation and avoids the challenge of distinguishing between objects in memory during run-time.

The last complication brought about by this modification of objects in memory is the use of complex objects from shared libraries. As these libraries, by definition, can be shared amongst multiple processes, the sharing of such a library containing complex objects could prove disastrous if shared by a process employing the unified approach and a process employing our proposed memory segregation approach as each process would expect different forms in memory of the shared complex objects. Fortunately, this too is a trivial matter. Just as modern 64-bit systems contain both 32-bit and 64-bit versions of various shared libraries in order to support the needs of their various processes, so too can systems contain unified and memory-segregated versions of their shared libraries that contain complex objects.

F. Parsing for Segregation

The task of the compiler to parse code to decipher where to allocate memory for data is already partially handled as modern compilers already parse code to decipher whether memory for a given piece of data should be allocated to a stack, heap, .bss section, or .data section. Thus, the only additional functionality required is that of identifying between non-control data, pointers, and control data related to the call stack so that memory for such forms of data is allocated not only to the correct data structure but to the correct memory segment as well. Identifying control data for the call stack is arguably the simplest of the three as their use is tightly coupled with the use of CALL, RET, and longjmp instructions as well as those instructions for interrupt and error handling. We believe distinguishing between program-defined pointers and non-control data is an equally trivial task in most instances, particularly with C-like programming languages in which pointers are designated in the language by one or more operators (e.g., *). Even with the use of memory allocation functions such as malloc and realloc, the task should not be overly complex as the use of these functions is often accompanied by an assignment to a typed value or a cast to a typed value. However, it must be noted that the use of void pointers without casts to typed values does increase the complexity, potentially resulting in forcing the compiler to allocate the data on the anonymous stack segment or anonymous data segment. We suspect that given sufficient static analysis capabilities, the compiler could yield a known type for these pointers, but that is a topic for future investigation.

IV. Combating Memory Corruption

To illustrate the merits of our proposed method of memory segmentation, we discuss its effects on the types of attacks used by the buffer overflow benchmark originally put forth by Wilander et al [20]. This benchmark uses two separate techniques, two separate buffer locations, and four attack targets, with two such targets existing in two forms, to create a total of 20 unique attacks. The two separate techniques used are to either *overflow the buffer all the way to the attack target* or *overflow the buffer to redirect a pointer to the target*. The two locations describe the positions in memory of the buffer that will be overflowed; either on *the stack or the heap/BSS/data segment*. The four targets used are the

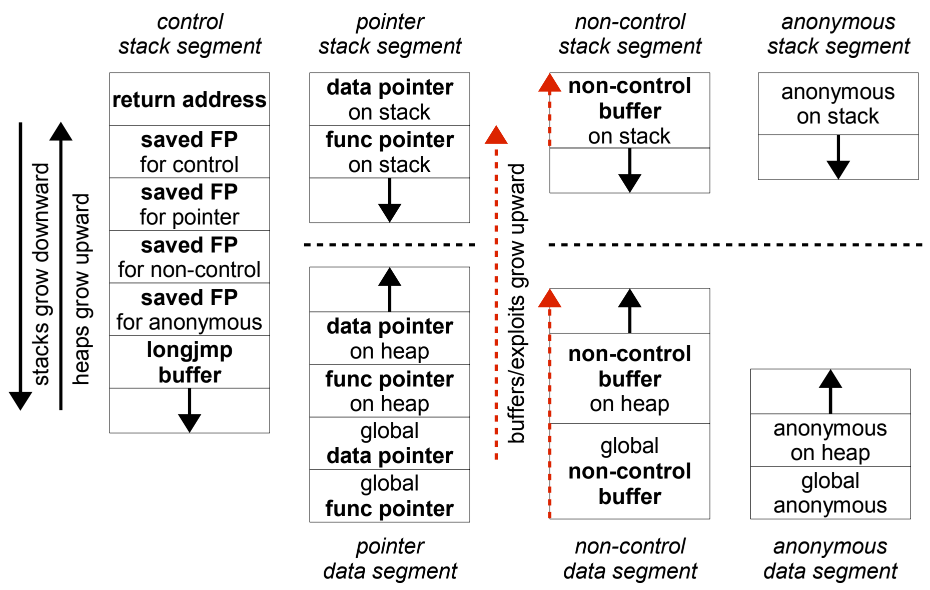

Fig. 4. Benchmark attacks in memory using our segregated stacks and heaps

return address, the saved frame pointer, function pointers, and longjmp buffers, with the longjmp buffers as either variables or function parameters.

As the buffers used in this benchmark are memory allocated to contain primitive datatypes, which are non-control data, under our proposed implementation the buffer in each attack would be allocated to the non-control stack segment or non-control data segment. Furthermore, as the targets are return addresses, saved frame pointers, function pointers, and longjmp buffers, under our proposed implementation, the target in each attack would be allocated to the control stack segment, pointer stack segment, or pointer data segment. Similarly, for the attacks in which the technique is to *overflow the buffer to redirect a point to the target*, any such pointer would be allocated to the pointer stack segment or pointer data segment. Thus, as is illustrated in Figure 4, the attacks employed by this benchmark would be blocked by our proposed memory segmentation as the buffer overflows occur in memory segments separate from those of their attack targets and any pointers they could use to reference those attack targets, preventing them from accessing those targets or pointers with the overflows of their buffers in the non-control stack segment and non-control data segment.

We believe this provides an accurate assessment of the impact of our proposed implementation on buffer overflow attacks on production systems as memory buffers are most often allocated to contain primitive datatypes; the use of a buffer allocated for program-defined pointers, return addresses, saved frame pointers or other control data is unusual (if not nonexistant) in production systems. It should be further emphasized that not only would our proposed memory segmentation mitigate these attacks, but they would prevent the corruption of control data by these attacks — restricting all corruption to non-control data. Thus, with no control data corrupted, it is more likely for an attacked process to continue functioning instead of resulting in a crash and fresh restart. Thus, given adequate functionality to handle the level of corruption of non-control data, the process could proceed as if merely given invalid inputs. It could even detect and alarm on the attempt to corrupt its control flow!

V. Limitations

For the sake of completeness, we will discuss the limitations of our proposed method of memory segmentation. First, self-modifying code contains instructions in which memory allocated to non-control data is written directly to memory containing instructions, control stack data, and/or pointers; such code blurs the lines between control and non-control data and as such our method would be ill-suited for protecting such code. Second, while our method prevents the direct corruption of control data, such as pointers, by non-control data overflows, it does *not* prevent the corruption of non-control data used as offsets of pointers used for indirection. As such, it may be possible for an attacker to corrupt such an offset to corrupt an address resulting from the calculation of the pointer and corrupt offset. Furthermore, as the current memory segmentation technology on some architectures relies on logical addresses provided from memory to select which memory segment to use for memory accesses, an attack could result in an alteration that would result in use of an unexpected memory segment, thus potentially changing a write intended for non-control data to instead write to control data. Fortunately, address space layout randomization [15] could be combined with our proposed memory segregation to aid in preventing this type of attack, although it would likely cause such an attack to result in a segmentation fault, forcing the process to crash. Third, as our method makes no claims as to the protection of non-control data, it is unable to protect

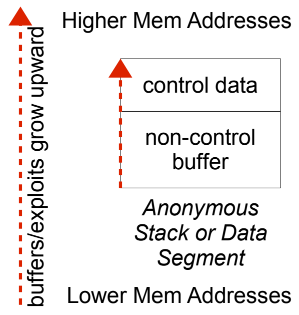

Fig. 5. Memory exploit enabled through the presence of control data and non-control data allocations as anonymous

against non-control data attacks [13]. Fourth, our method is largely dependent on the success of the compiler in parsing from code the forms of data used to allocate them to the correct memory segments. As the inability to decipher the form of a particular instance of data when allocating memory for it results in its allocation to either the anonymous stack segment or anonymous data segment, if such was to happen to both an instance of control data and an exploitable non-control data buffer such that the buffer appeared in lower memory than the control data, as illustrated in Figure 5, then that buffer could be overflowed to corrupt that control data. However, given a small enough number of anonymous instances of data, this could be prevented by allocated each instance to its own memory segment instead of using the anonymous stack segment and anonymous data segment. Last of all, our method outlined here is, at least in this form, dependent on a particular hardware and software platform combination.

VI. Future Work

The most logical place for future work is the completion of a working prototype. As previously discussed, this involves adjustments to the compiler. In addition, as was previously mentioned in our limitations, the corruption of an offset could potentially affect the selection of the memory segment used for a given memory access as this selection is dependent on the logical address supplied. To remedy this situation, it may be possible to handle the selection of memory segments in another manner, possibly, with hardware support, using only the currently executing instruction. Lastly, our proposed method for memory segregation shows potential for mitigating buffer overflows by restricting what they can corrupt and potentially mitigates these attacks without requiring the corrupted process to terminate. As such, additional research is warranted in other methods that may be combined with ours to further reduce the corruption possible from buffer overflows to not only control data but non-control data as well. We believe that by reducing the corruption possible from a buffer overflow we can increase the probability of a process successfully recovering from buffer overflow attacks, thus mitigating the use of buffer overflows for DoS attacks.

VII. Conclusion

In this paper we presented a concept for not only preventing control hijacking via buffer overflow attacks but for preventing the actual corruption of control data targeted by buffer overflows. We discussed making modifications to the compiler, so that processes could be recompiled to use multiple additional memory segments used to allocate memory to multiple segregated stacks, heaps, .bss sections, and .data sections, thereby enabling the segregation of control and non-control data in memory. Using an existing buffer overflow benchmark, we illustrated that the use of memory segregation would be an effective method for preventing the corruption of control data and thus in preventing the control flow hijacking and DoS attacks from buffer overflows. Finally, we have shown that it is possible to mitigate buffer overflow attacks by limiting what can be corrupted by a buffer overflow.

References

[1] E. Spafford, *The Internet Worm Program: Analysis*, in *ACM SIGCOMM Computer Communication Review*, Jan. 1989.

[2] *Write-what-where condition*, ASDR TOC Vulnerabilities, OWASP ASDR Project, http://www.owasp.org/index.php/Write-what-where_condition. Last accessed Feb. 2011.

[3] Aleph One, *Smashing The Stack For Fun And Profit*, *Phrack Magazine*, Nov. 1996.

[4] c0ntex, *Bypassing non-executable-stack during exploitation using return-to-libc*, http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf. Last accessed Feb. 2011.

[5] c0ntex, *How to hijack the Global Offset Table with pointers for root shells*, www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf. Last accessed Feb. 2011.

[6] *2010 CWE/SANS Top 25 Most Dangerous Software Errors*, MITRE and the SANS Institute, http://cwe.mitre.org/top25/index.html. Last accessed Feb. 2011.

[7] *A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*, http://support.microsoft.com/kb/875352. Last accessed Feb. 2011.

[8] *Pax pagexec documentation*, http://pax.grsecurity.net/docs/pageexec.txt. Last accessed Feb. 2011.

[9] Intel Corporation, *Intel Architecture Software Developers Manual, Volume 3: System Programming Guide*, Intel Corp., 2006, Order Number 243192.

[10] *Buffer overflow attacks bypassing dep (nx/xd bits) - part 2 : Code injection*, http://www.mastropaolo.com/?p=13. Last accessed Feb. 2011.

[11] Silberschatz, Galvin, and Gagne, *Operating System Concepts*, 8th Edition.

[12] J. Pincus and B. Baker, *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*, http://www.soe.ucsc.edu/classes/cmps223/Spring09/Pincus%2004.pdf. Last accessed Feb. 2011.

[13] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, *Non-Control-Data Attacks Are Realistic Threats*, in *Proc. of the 14th USENIX Security Symposium*, Aug. 2005, pages 177–192.

[14] klog, *The Frame Pointer Overwrite*, in *Phrack Magazine*, Sept. 1999.

[15] *Pax aslr documentation*, http://pax.grsecurity.net/docs/aslr.txt. Last accessed Feb. 2011.

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

[17] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, *Pointerguard: protecting pointers from buffer overflow vulnerabilities*, in *Proc. of the 12th USENIX Security Symposium*, Aug. 2003, pages 91–104.

[18] Vendicator, *Stack Shield: A "stack smashing" technique protection tool for Linux*, http://www.angelfire.com/sk/stackshield/info.html. Last accessed Feb. 2011.

[19] H. Etoh, *GCC extension for protecting applications from stack-smashing attacks*. Last accessed Feb. 2011.

[20] J. Wilander and M. Kamkar, *A comparison of publicly available tools for dynamic buffer overflow preventing*, in *Proceedings of the 10th Network and Distributed System Security Symposium*, Feb. 2003, pages 149–162.

[21] C. Cowan, *Re; PointerGuard: It's not the Size of the Buffer, it's the Address*, http://www.securityfocus.com/archive/1/333988. Last accessed Feb. 2011.

[22] S. Alexander, *defeating compiler-level buffer overflow protection*, in *;login: The USENIX Magazine*, June 2005.

[23] R. Riley, X. Jiang, and D. Xu, *An Architectural Approach to Preventing Code Injection Attacks*, in *IEEE Transactions on Dependable and Secure Computing*, Oct. 2010, pages 351–365.

[24] Tilo Mller, *ASLR Smack & Laugh Reference*, http://www.ece.cmu.edu/~dbrumley/courses/18739c-s11/docs/aslr.pdf. Last accessed Feb. 2011.

[25] Tyler Durden, *Bypassing PaX ASLR protection*, in *Phrack Magazine*, July 2002.

[26] J. Xu, Z. Kalbarxzyk, S. Patel, and R. K. Iyer, *Architecture Support for Defending Against Buffer Overflow Attacks*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.7372&rep=rep1&type=pdf, 2002. Last accessed Feb. 2011.

[27] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, *A Processor Architecture Defense Against Buffer Overflow Attacks*, in *IEEE International Conference on Information Technology: Research and Education*, 2003, pages 243–250.

[28] M. R. Krishnan, *Heap: Pleasures and Pains*, http://msdn.microsoft.com/en-us/library/ms810466.aspx. Last accessed Feb. 2011.

[29] *Malloc Multiheap*, http://publib.boulder.ibm.com/infocenter/aix/v6r1/topic/com.ibm.aix.genprogc/doc/genprogc/malloc_multiheap.htm. Last accessed Feb. 2011.

[30] *mmalloc.info*, http://www.slac.stanford.edu/comp/unix/package/rtems/doc/html/mmalloc/mmalloc.info.Top.html. Last accessed Feb. 2011.