IBM®

developerWorks.

LoP/Cell/B.E.: Buffer overflow vulnerabilities, Part 1: Understanding buffer overflow issues for Linux on Power-based systems

Skill Level: Intermediate

Ramon de Carvalho Valle (rcvalle@br.ibm.com)
Software Engineer
IBM

06 Jan 2009

Get acquainted with buffer overflow vulnerabilities in Linux® running on Power™/Cell Broadband Engine™ Architecture processor-based servers. Buffer overflows occur when a process tries to store data outside of the bounds of a fixed-length buffer. When that happens, all sorts of erratic system behavior can result, and some can be detrimental to your system's security. Part 1 of this article series briefly discusses buffer overflows and the Power and Cell/B.E.™ architectures, and then shows how you can change the process-execution flow in the target systems and overwrite a local variable in 32- and 64-bit modes. (Part 2 will show how to overwrite a function pointer in 32- and 64-bit modes and illustrate assembly components through shell, network, and socket code samples.)

In this article, all examples of buffer overflow vulnerabilities in Linux running on Power/Cell Broadband Engine Architecture processor-based servers were developed and executed on an IBM BladeCenter® JS22 Express server, an IBM BladeCenter QS21 server, and a Sony Playstation 3 running Red Hat Enterprise Linux 4 Update 7.

Review of buffer overflows

Let's start with a quick review of buffer overflows. A **buffer overflow**, or **buffer overrun**, occurs when a process attempts to store data beyond the boundaries of a fixed-length buffer. The result is that the extra data overwrites adjacent memory locations. The overwritten data can include other buffers, variables, program flow

© Copyright IBM Corporation 2008. All rights reserved.

data, etc. Overwriting this data can cause such problems as erratic program behavior, memory-access exceptions, program terminations of the crash variety, the wrong returned results, or the most dangerous thing for systems integrity: a breach of security.

Buffer overflows cause many software weaknesses and, therefore, are the basis of malicious exploits. C/C++ systems especially prone to overflows. They provide no built-in protection to stop accessing or overwriting data in any part of memory, and they don't automatically check that data written to a built-in buffer array is within the boundaries of that array. That's why you should always support a system that does bounds checking, either by you or by the compiler and runtime.

To learn more about buffer overflows and how to avoid them, read the developerWorks article "Secure programmer: Countering buffer overflows."

Power Architecture

The POWER (Performance Optimization With Enhanced RISC) Architecture, originally developed by IBM, was introduced with the RISC System/6000 product family in early 1990. In 1991, Apple, IBM, and Motorola, known as the AIM alliance, began the collaboration to evolve to the PowerPC® Architecture, expanding the architecture's applicability. In 1997, Motorola and IBM began another collaboration focused on optimizing PowerPC for embedded systems. At the end of 2004, the Power.org consortium was launched with the goal of developing community specifications and supporting development tools that work together to facilitate integration and enhanced implementations focused on the Power Architecture.

The Power Architecture is an open architecture defined by the Power Instruction Set Architecture (Power ISA) maintained by the Power Architecture Advisory Council, which ensures compatibility among implementations and allows anyone to design and fabricate Power Architecture-compliant processors. The Xbox 360 processor and the Cell Broadband Engine processor both shine as excellent examples.

A processor implementation that conforms to the Power Architecture has four basic classes of instructions:

- Branch instructions
- Fixed-point instructions and other instructions that use the fixed-point registers
- Floating-point instructions and decimal floating-point instructions
- Vector instructions

Fixed-point instructions operate on byte, halfword, word, and doubleword operands.

© Copyright IBM Corporation 2008. All rights reserved.

Floating-point instructions operate on single-precision and double-precision floating-point operands. Vector instructions operate on vectors of scalar quantities and on scalar quantities, where the scalar size is byte, halfword, word, and quadword. The processor uses instructions that are four bytes long and word-aligned. It provides for byte, halfword, word, and doubleword operand fetches and stores between storage and a set of 32 General Purpose Registers (GPRs). It provides for word and doubleword operand fetches and stores between storage and a set of 32 Floating-Point Registers (FPRs). It also provides for byte, halfword, word, and quadword operand fetches and stores between storage and a set of 32 Vector Registers (VRs).

- The Condition Register (CR) is a 32-bit register that reflects the result of certain operations and provides a mechanism for testing (and branching).
- The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the Branch Conditional to Link Register instruction, and it holds the return address after Branch instructions.
- The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions.
- The Machine State Register (MSR) is a 64-bit register. This register defines the state of the processor. The 64th bit defines whether the processor is in 32-bit or 64-bit mode (0 or 1).

Processors provide two execution modes: 64-bit and 32-bit mode. In both modes, instructions that set a 64-bit register affect all 64 bits. The computational mode controls how the effective address is interpreted, how status bits are set, how the Link Register is set by Branch instructions, and how the Count Register is tested by Branch Conditional instructions. Nearly all instructions are available in both modes. In both modes, effective address computations use all 64 bits of the relevant registers (GPRs, LRs, CTRs, etc.) and produce a 64-bit result. However, in 32-bit mode, the high-order 32 bits of the computed effective address are ignored for the purpose of addressing storage.

All instructions are four bytes long and word-aligned. Thus, whenever instruction addresses are presented to the processor (as in Branch instructions), the low-order two bits are ignored. Similarly, whenever the processor develops an instruction address, the low-order two bits are zero.

Bits 0:5 always specify the opcode. Many instructions also have an extended opcode. The remaining bits of the instruction contain one or more fields for the different instruction formats.

A program references storage using the effective address computed by the processor when it executes a Storage Access or Branch instruction or when it

© Copyright IBM Corporation 2008. All rights reserved.

fetches the next sequential instruction. Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte. The byte ordering (Big-Endian or Little-Endian) for a storage access is specified by the operating system.

Cell Broadband Engine Architecture (CBEA)

The Cell Broadband Engine (Cell/B.E.) processor is the first implementation of a new multiprocessor family conforming to the Cell Broadband Engine Architecture (CBEA). The CBEA is an architecture that extends the 64-bit Power Architecture. The CBEA and the Cell/B.E. processor are the result of a collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001.

Although the Cell/B.E. processor was initially intended for applications in media-rich consumer-electronics devices, such as game consoles and high-definition televisions, the architecture is designed to enable fundamental advances in processor performance. These advances are expected to support a broad range of applications in both commercial and scientific fields.

The most distinguishing feature of the Cell/B.E. processor is that, although all processor elements share memory, their function is specialized into two types: the Power Processor Element (PPE) and the Synergistic Processor Element (SPE). The processor has one PPE and eight SPEs.

The first type of processor element, the PPE, contains a 64-bit Power Architecture core. It complies with the 64-bit Power Architecture and can run 32-bit and 64-bit operating systems and applications.

The second type of processor element, the SPE, is optimized for running compute-intensive SIMD applications; it is not optimized for running an operating system. The SPEs are independent processor elements, each running their own individual application programs or threads. Each SPE has full access to coherent shared memory, including the memory-mapped I/O space.

There is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system and, in many cases, the top-level thread control for an application. The PPE depends on the SPEs to provide the bulk of the application performance.

The most significant difference between the SPE and PPE lies in how they access memory. The PPE accesses main storage (the effective-address space) with load and store instructions that move data between main storage and a private register file, the contents of which may be cached.

The SPEs access main storage with direct memory access (DMA) commands that

© Copyright IBM Corporation 2008. All rights reserved.

move data and instructions between main storage and a private local memory, called a local store or local storage (LS). An SPE's instruction-fetches and load and store instructions access its private LS rather than shared main storage; the LS has no associated cache. This three-level organization of storage (register file, LS, main storage), with asynchronous DMA transfers between LS and main storage, is a radical break from conventional architecture and programming models because it explicitly parallelizes computation with the transfers of data and instructions that feed computation and store the results of computation in main storage.

Controlling buffer overflows

Now we'll look at:

- Changing the process-execution flow
- Overwriting a local variable in 32-bit mode
- Overwriting a local variable in 64-bit mode

(And in Part 2, we'll cover overwriting a function pointer in both 32- and 64-bit modes and provide some example code.)

Changing process execution flow

Similarly to the x86/x86_64 architectures, a given process's execution flow in Power/CBEA can be changed by the following:

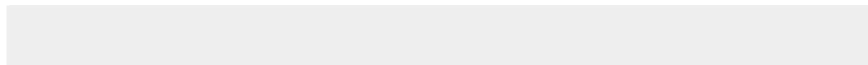
- Overwriting a local variable that is near the buffer in memory of a given process's virtual address space to change the behavior of the application.
- Overwriting the saved return-instruction pointer in a stack frame. Upon returning from the called function, execution will resume at the saved return-instruction pointer as specified.
- Overwriting a function pointer or exception handler that is subsequently executed.

Overwriting a local variable in 32-bit mode

This section discusses how a given process's execution flow can be changed by overwriting a local variable.

The following example is vulnerable to a heap-based buffer overflow:

Listing 1. example1.c (vulnerable to a heap-based buffer overflow)



© Copyright IBM Corporation 2008. All rights reserved.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct mystruct {
    unsigned char buffer[16];
    unsigned long cookie;
};

int
main(int argc, char **argv)
{
    struct mystruct *s;

    if ((s = malloc(sizeof(struct mystruct))) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    s->cookie = 0;

    if (argc > 1)
        strcpy(s->buffer, argv[1]);

    if (s->cookie == 0x42424242) {
        printf("Congratulations! You won a cookie!\n");
        exit(EXIT_SUCCESS);
    }

    printf("Hello world!\n");

    exit(EXIT_SUCCESS);
}
```

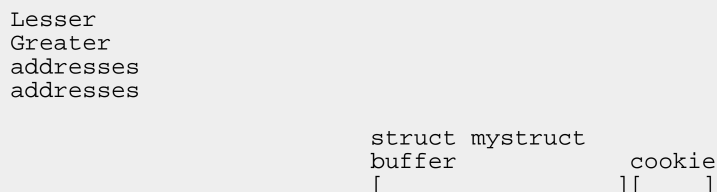
Listing 1 does not validate user-supplied data when copying it to the `buffer` member of the previously allocated `struct mystruct` using the `strcpy` function, resulting in a heap-based buffer overflow. Normal execution of the example writes the *"Hello world!"* string to `stdout`.

Listing 2. Compilation and execution of Listing 1

```
$ gcc -Wall -o example1 example1.c
$ ./example1
Hello world!
$
```

Figure 1 represents the `struct mystruct` and its members in the heap segment.

Figure 1. The struct mystruct



© Copyright IBM Corporation 2008. All rights reserved.

```
Bottom of
Top of
heap
heap
```

The process's execution flow can be changed by overwriting the `cookie` member of `struct mystruct` that is located right after the buffer in memory with the `0x42424242` value (BBBB in the ASCII character set).

Listing 3. Overwriting the cookie member

```
$ ./example1 AAAAAAAAAAAAAAAABBBB
Congratulations! You won a cookie!
$
```

Figure 2 represents the `struct mystruct` and its members in the heap segment after the overflow.

Figure 2. The struct mystruct after the overflow

```
Lesser
Greater
addresses
addresses
                         struct mystruct
                         buffer           cookie
                         [AAAAAAAAAAAAAAAA][BBBB]

Bottom of
Top of
heap
heap
```

Overwriting a local variable in 64-bit mode

Now let's talk about how a given process's execution flow can be changed by overwriting a local variable in 64-bit mode. In the C language, only long and pointer data types are changed between 32-bit and 64-bit modes. Any pointer arithmetic should be performed using variables of type long regardless if in 32-bit or 64-bit mode. Pointer assignment should only be performed between other pointers or variables of type long.

The following example is vulnerable to a heap-based buffer overflow:

Listing 4. example2.c (vulnerable to a heap-based buffer overflow)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```


© Copyright IBM Corporation 2008. All rights reserved.

```
struct mystruct {
    unsigned char buffer[16];
    unsigned long cookie;
};

int
main(int argc, char **argv)
{
    struct mystruct *s;

    if ((s = malloc(sizeof(struct mystruct))) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    s->cookie = 0;

    if (argc > 1)
        strcpy(s->buffer, argv[1]);

    if (s->cookie == 0x4242424242424242) {
        printf("Congratulations! You won a cookie!\n");
        exit(EXIT_SUCCESS);
    }

    printf("Hello world!\n");

    exit(EXIT_SUCCESS);
}
```

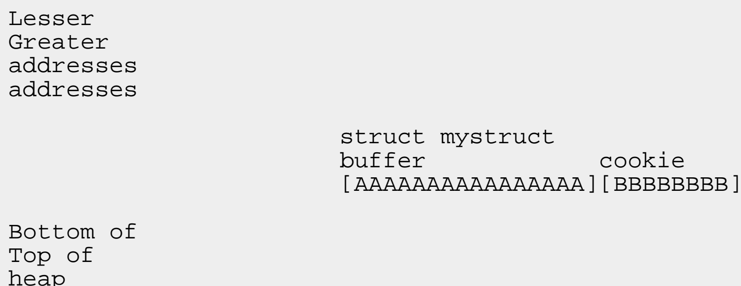
The process's execution flow can be changed by overwriting the `cookie` member of `struct mystruct` that is located right after the buffer in memory with the `0x4242424242424242` value (BBBBBBBB in the ASCII character set).

Listing 5. Overwriting the cookie member

```
$ gcc -Wall -m64 -o example2 example2.c
$ ./example2 AAAAAAAAAAAAAAAABBBBBBBB
Congratulations! You won a cookie!
$
```

Figure 3 represents the `struct mystruct` and its members in the heap segment after the overflow.

Figure 3. The struct mystruct after the overflow



© Copyright IBM Corporation 2008. All rights reserved.

```
heap
```

In the next installment

We've just scratched the surface. In Part 2, I'll show how to overwrite a function pointer and cover assembly components and some juicy shell, network, socket code samples.

© Copyright IBM Corporation 2008. All rights reserved.

Resources

Learn

- "POWER to the people" (developerWorks, December 2005) is a history of chipmaking at IBM.
- "PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors" (IBM Technical Library, February 2000) is a manual is to help programmers provide software that is compatible across the family of 32-bit PowerPC processors.
- "The Programming Environments Manual for 64-bit Microprocessors" (IBM Technical Library, July 2005) describes features common to 64-bit PPC processors and indicates which features are optional or may be implemented differently in the design of each processor.
- "Developing Embedded Software For The IBM PowerPC 970FX Processor" (IBM Technical Library, July 2004) is an application note that discusses issues associated with developing new software and porting existing software to the PowerPC 970FX processor, including an overview of the PowerPC 64-bit architecture as implemented in the PowerPC 970FX processor.
- "Cell Broadband Engine Programming Handbook" (IBM Technical Library, April 2007) provides information for developing applications, libraries, middleware, drivers, compilers, or operating systems for the Cell Broadband Engine.
- "Power Instruction Set Architecture Version 2.05" (Power.org, October 2007; PDF document) is the instruction set specification.
- The UNIX Assembly Components for Proof of Concept Codes project contains a set of assembly components for proof-of-concept codes on different operating systems and architectures.
- "Secure programmer: Countering buffer overflows" (developerWorks, January 2004) presents the top vulnerability in Linux/UNIX systems—buffer overflows—by explaining what they are, why they're common, why they're dangerous, and how to counter them.
- In the developerWorks Linux zone, find more resources for Linux developers (including developers who are new to Linux), and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks,

© Copyright IBM Corporation 2008. All rights reserved.

build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and spaces.

About the author

Ramon de Carvalho Valle
Ramon is a Software Engineer at the IBM Linux Technology Center in Sao Paulo, Brazil. He is a Founder/Security Researcher at RISE Security and has extensive experience in vulnerability research, exploitation techniques, exploit development, and reverse engineering on a wide range of operating systems and architectures. He also contributes to open source projects like The Metasploit Framework.

Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of IBM trademarks.
Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.
Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

© Copyright IBM Corporation 2008. All rights reserved.