Architectural Support for Mitigating Buffer-Overflow Software Attacks

Love Kumar Sah and Srinivas Katkoori
Computer Science and Engineering, University of South Florida
Email: {lsah, katkoori}@mail.usf.edu

1. Abstract

Significant work on vulnerabilities focuses on buffer overflows, in which data exceeding the bounds of an array is loaded into the array. The loading continues past the array boundary, causing variables and state information located adjacent to the array to change. As the process is not programmed to check for these additional changes, the process acts incorrectly. The incorrect action often places the system in a non-secure state. In this poster, we present a novel hardware approach to detect buffer overflow problem through variable base and bound checking during run time.

2. Buffer Overflow Attack

A buffer overflow happens when data written to a buffer overflows into memory adjacent to the allocated buffer. Taking advantage of corruption of the adjacent memory is the basis of the attack.

```
char x[6];
char y[4] = "0009";
char temp[ ] = "attackers\0"
strcpy(x,temp );
```

To illustrate a basic buffer overflow [1], consider the c code above which have of a 6-byte long string buffer X followed by a 4-byte long integer Y shown in Fig.1 Initially, the string A contains only "-" characters and the integer B contains the number 9.

X	X	X	X	X	X	Y	Y	Y	Y
-	-	-	-	-	-	0	0	0	9

(a)

X	X	X	X	X	X	X	X	Y	Y
a	t	t	a	c	k	e	r	s	\0

(b)

Fig.1. (a) Memory content of adjacent array inside stack before strcpy() call. (b) Memory content after function call.

If a program stuffs too big a word into buffer X, the result is an overflow into Y. For example store the string "attackers" into X. (Assume that the string is terminated with a null byte: '\0').

Rather than relying on individual skill to write buffer-overflow free program, a variety of methods to protect systems from buffer-overflow attacks have been proposed. Fig. 2 shows the taxonomy of buffer overflow protection classification

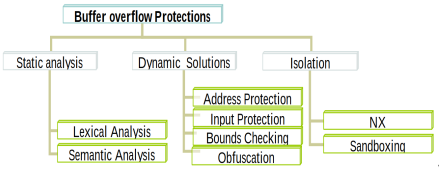

Fig. 2. Buffer overflow protection classification [1]

Full protection is not completely provided by most of them. As protection has improved, attacks have adapted, so an "arms race" developed.

3. Bound Checking

Base and Bound information helps to quickly check if the presented address is a valid to access. Bound checking is inexpensive to add and provide excellent memory protection mechanism. One form of bound checking can be seen in multiprocessing operation system which check the memory access to its designated location only.

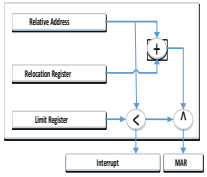

Fig. 3. CPU address bound checking hardware

4. Proposed Architectural Support

Named variables in the program can be of three types, namely (a) scalar variables, (b) arrays, and (c) pointers. Meta data like base and bound information of array and pointer variable is important to determine the attack condition. We propose these information to stored at the beginning of the variable address along with its data as shown in Fig. 3. Whenever the variable access happens, It access this data along with base and bound value to confirm access is within its bound.

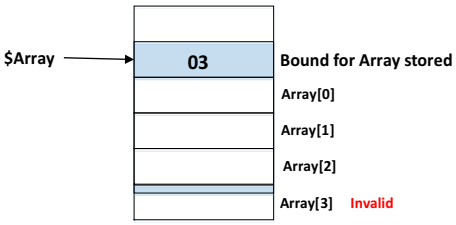

Fig 4. Memory layout for a variable

In proposed pipelined architecture (Fig. 5), all load (lw) and store (sw) instructions will check the base value of the register which stores address of variable metadata (Fig. 4). Comparing this data with offset will allow us to determine if it is a valid access or not.

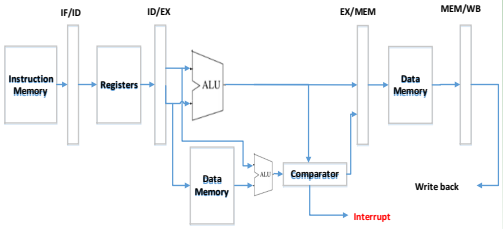

Fig. 5. Proposed pipeline architecture for the bound check during load and store instruction execution.

5. Proposed Methodology

During the compilation every named variable in the program is augmented with meta data that specifies its legal boundaries. For bounds checking in pointers, the compiler uses the meta data of the object it is dereferencing rather than the pointer's own meta data At runtime, all load and store instructions using these variables are routed through the test shown in Fig. 5. The hardware will perform bound checks and will interrupt if fails.

For every named variable in the program, the compiler creates two additional fields for storing base and bound data. These fields are created at offsets −4 and −8 bytes with respect to the variable shown in Fig. 6.

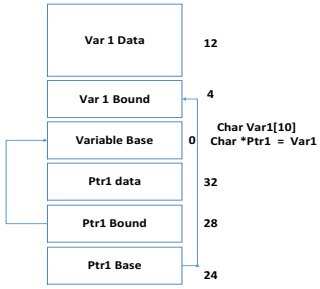

Fig. 6. Memory organization for variables

6. Experimental Results

Modified Simplescalar simulator [6] was used to test several programs from three benchmark suites. Simplescalar implements PISA architecture which is basically a RISC architecture..

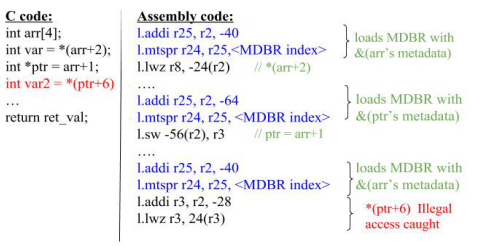

Fig. 7. Base and bound set before each lw and sw instruction.

We insert .asm code against a variable in the test c program as shown in Fig. 7. This code set the base and bound of a variable to a specific memory location. The modified Simplescalar sim-cache simulator detect this base and bound value against test variable register during runtime. Three widely-used open-source programs were chosen to test the effectiveness of the static analysis tools: BIND, WU-FTPD, and Sendmail. BIND is the most popular DNS server, WU-FTPD is a popular FTP daemon, and Sendmail is the dominant mail transfer agent (MTA). Moreover we tested the correctness of the implementation by running several exploit code. We tested our implementation on a corpus of 291 small C program test cases [7]. Table 1 shows the overhead of the program before and after change in simulator program. The overhead is because of load store instruction associated additional code.

Benchmark		Number of Instructions		Overhead %
		Original Program	Protected Program	
Bind		12412	13326	7.36
Sendmail		14562	15458	6.15
Wu-ftpd		8458	9045	6.94
NIST	Ahdec1-bad	9451	10124	8.07
	Ahgets1-bad	19562	21136	8.04

Table 1. Simulation results of benchmarks

7. Conclusion

Buffer overflow is one of the top vulnerabilities in modern era. Even secure programming language such as Java suffers this problem. Software approach along with compiler support is not a perfect solution for this problem. We have successfully demonstrated that the architectural support can aid in detecting software overflow attacks during runtime. The associated overhead in term of additional instruction is minimal.

8. References

1. K. Piromsopa and R. Enbody, "Survey of protections from bufferoverflow attacks," Engineering Journal, vol. 15, 2011.
3. Z. Shao and B. Xiao, "Efficient Array & Pointer Bound Checking Against Buffer Overflow Attacks via Hardware/Software," Proceedings of the International Conference on Information Technology: Coding and Computing, Nevada, 2005.
4. G.Krishnakumar and K.Veezhinathan "GANDALF: A Fine-Grained Hardware-Software Co-Design For Preventing Memory Attacks ," IEEE Embedded Systems Letters, 2018.
6. http://www.simplescalar.com
7. https://www.ll.mit.edu/mission/cybersec/corpora/Cyberrpora/Cybersystemscorpora.html