International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Automatic Patch Generation for Control Hijacking Attacks

Saud Adam Abdulkadir[1], Savaridassan P.[2]

[1]**Research Scholar,** Department of Information Technology, Faculty of Engineering and Technology, SRM University, Kattankulathur, Chennai, India

[2]**Assistant Professor,** Department of Information Technology, Faculty of Engineering and Technology SRM University, Kattankulathur, Chennai, India

Abstract: ***With the number of solutions proposed to provide an adequate protection against overflow attacks, integer overflow still pose a significant threat to security and availability of today's computing environments. Most of the existing solutions terminate the vulnerable program when the overflow occurs, rendering the program unavailable and leading to denial of service. The impact on system availability is a serious problem on service orientated platforms. We will provide a solution that will automatically diagnose and patch any suspicious integer input that will lead to overflowing the values and causing further threats to possible buffer overflow in the system. The key idea of our solution is to virtualized memory accesses and moves the vulnerable buffer into protected memory regions, which provides a fundamental and effective protection against recurrence of the same attack without stopping the normal system execution.***

Keywords: Software Security, Integer Overflow Attacks Prevention, Vulnerability Patching.

1. Introduction

As the demand for high availability of services continue to grow significantly, with the fast development of service-oriented computing paradigm, system security and availability is still severely hindered by control hijacking attacks.

This type of attacks typically compromise a control sensitive data structure in a vulnerable system, among the vulnerabilities is the integer overflow, buffer overflow and format string [1].

An integer overflow condition exists when an integer, which has not been properly sanity checked, is used in the determination of an offset or size for memory allocation, copying, concatenation, or similarly. If the integer in question is incremented past the maximum possible value, it may wrap to become a very small or negative number, therefore providing a very incorrect value.

Integer overflows are for the most part only problematic in that they lead to issues of availability (DoS). Common instances of this can be found when primitives subject to overflow are used as a loop index variable.

In some situations, however, it is possible that an integer overflow may lead to an exploitable buffer overflow condition. In these circumstances, it may be possible for the attacker to control the size of the buffer as well as the execution of the program.

Recently, a number of integer overflow-based, buffer-overflow conditions have surfaced in prominent software packages. Due to this fact, the relatively difficult to exploit condition is now more well-known and therefore more likely to be attacked. The best strategy for mitigation includes: a multi-level strategy including the strict definition of proper behavior (to restrict scale, and therefore prevent integer overflows long before they occur); frequent sanity checks; preferably at the object level; and standard buffer overflow mitigation techniques.

Not accounting for integer overflow can result in logic errors or buffer overflow. Integer overflow errors occur when a program fails to account for the fact that an arithmetic operation can result in a quantity either greater than a data type's maximum value or less than its minimum value. These errors often cause problems in memory allocation functions, where user input intersects with an implicit conversion between signed and unsigned values. If an attacker can cause the program to under-allocate memory or interpret a signed value as an unsigned value in a memory operation, the program may be vulnerable to a buffer overflow [2].

In the real world, 255 + 1 = 256. But to a computer program, sometimes 255 + 1 = 0, or 0 – 1 = 65535, or may be 40,000 + 40,000 = 14464. You don't have to be a math whiz to smell something fishy. Actually, this kind of behavior has been going on for decades, and there's a perfectly rational and incredibly boring explanation. Ultimately, it's buried deep in the DNA of computers, who can't count to infinity even if it sometimes feels like they take that long to complete an important task. When programmers forget that computers don't do math like people, bad things ensue - anywhere from crashes, faulty price calculations, infinite loops, and execution of code [3].

It is important to identify integer overflows before an attacker does. Given program source code, there are several techniques and tools (e.g., RICH [4], EXE [5], KLEE [6]) that can perform static analysis or model checking to detect integer overflows. However, as source code is not always available to end users, the state-of-the-art techniques have to rely on dynamically running the software, exploring program paths (e.g., SAGE [7]), and generating test cases, to show the

existence of a vulnerability. Such fuzzing techniques have been commonly used by underground attackers [8].

Consequences

- **Availability:** integer overflow generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likely hood of the infinite loops is also high.
- **Integrity:** If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the integer overflow has resulted in a buffer overflow condition, data corruption will most likely take place.
- **Access control (instruction processing)**: Integer overflows can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy [2].

2. Overflow Condition in Database

When MySQL stores a value in a numeric column that is outside the permissible range of the column data type, the result depends on the SQL mode in effect at the time:

- If strict SQL mode is enabled, MySQL rejects the out-of-range value with an error, and the insert fails, in accordance with the SQL standard.
- If no restrictive modes are enabled, MySQL clips the value to the appropriate endpoint of the range and stores the resulting value instead.

When an out-of-range value is assigned to an integer column, MySQL stores the value representing the corresponding endpoint of the column data type range. If you store 256 into a TINYINT or TINYINT UNSIGNED column, MySQL stores 127 or 255, respectively.

When a floating-point or fixed-point column is assigned a value that exceeds the range implied by the specified (or default) precision and scale, MySQL stores the value representing the corresponding endpoint of that range.

Column-assignment conversions that occur due to clipping when MySQL is not operating in strict mode are reported as warnings for ALTER TABLE, LOAD DATA INFILE, UPDATE, and multiple-row INSERT statements.

In MySQL, overflow handling during numeric expression evaluation depends on the types of the operands:

- Integer overflow results in silent wraparound.
- DECIMAL overflow results in a truncated result and a warning.
- Floating-point overflow produces a NULL result.

For example, the largest signed BIGINT value is 9223372036854775807, so the following expression wraps around to the minimum BIGINT value:

```
   mysql>SELECT 9223372036854775807 + 1;
+-------------------------+
| 9223372036854775807 + 1 |
+-------------------------+
| -9223372036854775808 |
+-------------------------+
```

To enable the operation to succeed in this case, convert the value to unsigned;

```
mysql>SELECT CAST(9223372036854775807 AS UNSIGNED) + 1;
+--------------------------------------+
|CAST(9223372036854775807 AS UNSIGNED)+1|
+--------------------------------------+
|9223372036854775808 |
+--------------------------------------+
```

Whether overflow occurs depends on the range of the operands, so another way to handle the preceding expression is to use exact-value arithmetic because DECIMAL values have a larger range than integers:

```
mysql>SELECT 9223372036854775807.0 + 1;
+---------------------------+
| 9223372036854775807.0 + 1 |
+---------------------------+
| 9223372036854775808.0 |
+---------------------------+
```

Subtraction between integer values, where one is of type UNSIGNED, produces an unsigned result by default. If the result would otherwise have been negative, it becomes the maximum integer value. If the NO_UNSIGNED_SUBTRACTION SQL mode is enabled, the result is negative.

```
mysql>SET sql_mode = '';
mysql>SELECT CAST(0 AS UNSIGNED) - 1;
+--------------------------+
| CAST(0 AS UNSIGNED) - 1 |
+--------------------------+
| 18446744073709551615 |
+--------------------------+

mysql>SET sql_mode =
'NO_UNSIGNED_SUBTRACTION';
mysql>SELECT CAST(0 AS UNSIGNED) - 1;
+--------------------------+
| CAST(0 AS UNSIGNED) - 1 |
+--------------------------+
| -1 |
+--------------------------+
```

If the result of such an operation is used to update an UNSIGNED integer column, the result is clipped to the maximum value for the column type, or clipped to 0 if NO_UNSIGNED_SUBTRACTION is enabled. If strict SQL mode is enabled, an error occurs and the column remains unchanged [9].

3. Proposed System

We are going to implement a system that will use some ideas from SafeStack [10]. Where SafeStack uses Memory Access Visualization to relocate bug-triggering buffers, the system will also relocate the vulnerable buffers to safer locations in order to maintain program functionality at a runtime.

The system will consist of production system and patching system. The production system is the deployment of the application in the production environment, the patching system is the application which is used to identify the bug triggering buffers and generate patches.

We can move buffers to a monitored memory region to detect whether some of them have out-of-bound access. According to the buffer information, the system will move the buffer objects to the monitored memory areas, and identifies bug triggering buffers through testing. Once a bug-triggering buffer is detected, the patch generator generates runtime patches, which are evaluated by the patch evaluation component to determine whether they are able to make the application tolerate attack recurrences. If so, the patch evaluation component sends the generated runtime patches to protected memory area and then writes values into the corresponding protected memory area instead of the original address space.

Finally, the runtime patch applicator will applies patches generated from the system, in this way; the system temporarily fixes the vulnerability and prevents subsequent attacks of the same type, which helps the production system withstand the overflow attacks until the vendor's official patch becomes available.

4. Related Work

PASAN [1] instruments an application's source code with additional instructions to check the control-sensitive data structures and to record sufficiently detailed execution state log from which one can create an attack signature and patch for a detected attack later on. Therefore, after detecting an attack, PASAN's repair-time component first finds the corresponding target address that was compromised. Then it traces back through the dynamic data dependency graph from the compromised target address and eventually reaches certain bytes in some network packet(s).

StackGuard [11**]** enhanced programs and define the behavior of writing to the return address of a function while it is still active. StackGuard prevents changes to active return addresses by either detecting the change of the return address before the function returns, or by completely preventing the write to the return address. Detecting changes to the return address is a more efficient and portable technique, while preventing the change is more secure. StackGuard supports both techniques, as well as adaptively switching from one mode to the other.

DieHard[12] memory manager places objects randomly across a heap whose size is a multiple of the maximum required. The resulting spacing between objects makes it likely that buffer overflows end up overwriting only empty space. Randomized allocation also makes it unlikely that a newly-freed object will soon be overwritten by a subsequent allocation, thus avoiding dangling pointer errors.

5. Experiment

Our experimental platforms consists of mysql version 5.1.36, Netbeans used as an interface for interacting and managing the vulnerable database run on windows 8.
We experiment the system with an out-of- range input file and the result is as follows:

- When strict sql mode is enabled, mysql rejects the out-of-bound value with and error message which eventually caused a denial of service (DoS).
- When no restrictive modes are enabled, then an overflow condition occurs based on:
  a) Integer data type
     - Wraparound
  b) Decimal data type
     - Truncation
  c) Floating point
     - Null Result

6. Conclusion

Our system is designed to detect a vulnerable data that will overflow the data structure which will allow an adversary to take advantage of the weakness.

The system works in similar way with SafeStack, by identifying the overflowed structures in the attack and mitigates the attacks to preserve the system availability at a run time and prevent an undesirable situation for an important business application from running down.

References

[1] Smirnov, A. Stony Brook Univ., tony Brook Tzi-ckerChiueh PASAN: Automatic Patch Generation for Buffer Overflow Attacks.
[2] https://www.owasp.org
[3] http://cwe.mitre.org/top25/index.ht l#CWE-190
[4] D. Brumley, T. ckerChiueh, R. Johnson, H. Lin, and D. Song. Rich: Automatically protecting against integer-based vulnerabilities. In Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07), 2007.
[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In Proceedings of the 13th ACM conference on Computer and communications security (CCS'06), pages 322–335, 2006.
[6] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In USENIX Symposium on Operating Systems Design and Implementation (OSDI'08), San Diego, CA, 2008.
[7] P. Godefroid, M. Levin, and D. Molnar. Automated white box fuzz testing. In Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), San Diego, CA, February 2008.
[8] Tielei Wang, TaoWei, Zhiqiang Lin, Wei Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution.
[9] http://dev.mysql.com/doc/refman
[10] Gang Chen, Hai Jin, DeqingZou, Bing Bing Zhou, Zhenkai Liang, WeideZheng and Xuanhua Shi. SafeStack: Automatically Patching Stack-based Buffer Overflow Vulnerabilities.
[11] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks," in Proceedings of the 7th conference on USENIX Security Symposium. USENIX Association, 1998, pp. 63–78

[12] E. Berger and B. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," in Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation. ACM, 2006, pp. 158–168.