Integer overflow & Algorithmic complexity vulnerabilities

Lecture 18

Module questionnaire

- Please take a few minutes to complete the module questionnaire

INTEGER VULNERABILITIES

Numbers in a CPU

- Mismatch between machine arithmetic and mathematical arithmetic
- Integer representation in today's architecture
  - Two's complement
  - See section 4.2.1 of Intel Software Developer's Manual
  - Good introduction: http://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

Two's complement

Unsigned integers

- Ordinary binary values ranging from 0 to max positive number that can be encoded given the integer size (N)
- Range: $[0, 2^N-1]$
  - 4 bit: [0000, 1111]

Signed integers

- Two's complement representation
- Sign bit
- Range: $[-2^N, 2^N-1]$
  - 4 bit, [1000,0111]

Integer overflow/underflow

- Integer overflow/underflow
  - Arithmetic expression results in value that is larger/smaller than can be represented on machine type
  - Typically, result silently “wraps around”

```
char * vuln_alloc(int size, int n) {
  return (char *) malloc(size * n);
}
```

- If the multiplication overflows, the allocated memory may be smaller than expected, which may lead to a buffer overflow

Width conversion

- Insecure conversion of an integral type to a wider or narrower integral type which has different range of values

```
#include <sys/types.h>
void vuln_copy(int16_t n, char *p, char *q) {
  uint32_t m = n;
  memcpy(p, q, m);
}
```

Signed/Unsigned conversion

- Converting a signed integer type to an unsigned integer type of the *same width* can change a negative number to a large positive number (or vice versa)

```
void vuln_copy(int n, char *p, char *q) {
  if (n > 800) // too large
    return;
  memcpy(p, q, m);
}
```

n = -1; m = 4294967295

- If n is negative, it will pass the bound check, but will copy a large number of bytes when promoted to unsigned int

Quiz

- `short x = 0x7FFF; x++;`
  - What is the value of x?
- `unsigned long l; short x = -2; l = x;`
  - What is the value of l?

Exploitable integer overflows

```
int main(int argc, char *argv[])
{
  char buf[512];
  long max;
  short len;

  max = sizeof(buf);
  len = strlen(argv[1]);
  printf("max %ld len %hd\n", max, len);
  if (len < max) {
    strcpy(buf, argv[1]);
  }

  return 0;
}
```

Integer Overflows

```
$ ./integeroverflow `python -c 'print "A" * 0x7fff'`
max 512 len 32767

$ ./integeroverflow `python -c 'print "A" * 0x8000'`
max 512 len -32768
Segmentation fault (core dumped)
```

Binary search in JDK

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

Fix:

`low + ((high – low) / 2)`

Vulnerable?

Read more:
http://googleresearch.blogspot.co.uk/2006/06/extra-extra-read-all-about-it-nearly.html

From the research dept

- http://www.phrack.org/issues.html?issue=60&id=10
- D. Brumley et al., RICH: Automatically protecting against integer-based vulnerabilities, NDSS 2007
  - Formal typing rules for safe integer operations
  - Extend compiler to add dynamic checks when program violates type safety
- T. Wang et al., IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution, NDSS 2009
  - Symbolic execute of binary code to find integer overflow
- D. Molnar et al., Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs, USENIX Security 2009
  - SmartFuzz: tool to perform symbolic execution and dynamic test generation

Defenses

- Manual defenses
  - Avoid mixing signed/unsigned
  - Avoid mixing different width
  - Check for overflow/underflow
- SafeInt
  - Used extensively throughout Microsoft
  - http://safeint.codeplex.com/

ALGORITHMIC COMPLEXITY VULNERABILITIES

Denial of service attacks

- DoS attacks are based on the idea of overwhelming the target with the sheer number of requests
  - “smurf” ICMP attack
  - DNS amplification
- Other approach: are there special requests that cause the target to perform a disproportionate amount of work?
- Direction: look for requests that cause typically efficient algorithms to perform badly
  - Algorithmic complexity

Algorithmic complexity

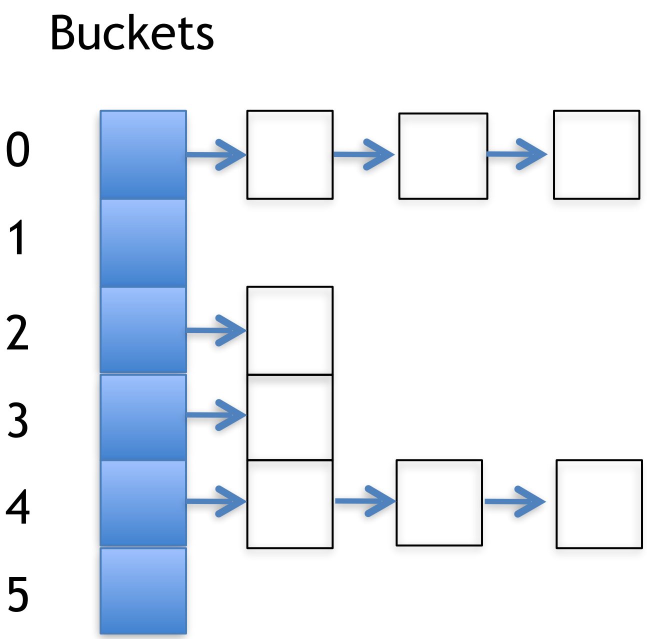
- Data structures frequently used in applications have “average-case” expected running time that is far more efficient than worst case
  - They are used because typical/benign inputs do not expose the worse case
- Examples
  - Hash tables can degenerate to linked lists

    $O(1) \rightarrow O(n)$ insertion/lookup time of one element ($O(n^2)$ for n element)
  - Binary trees can degenerate to linked lists

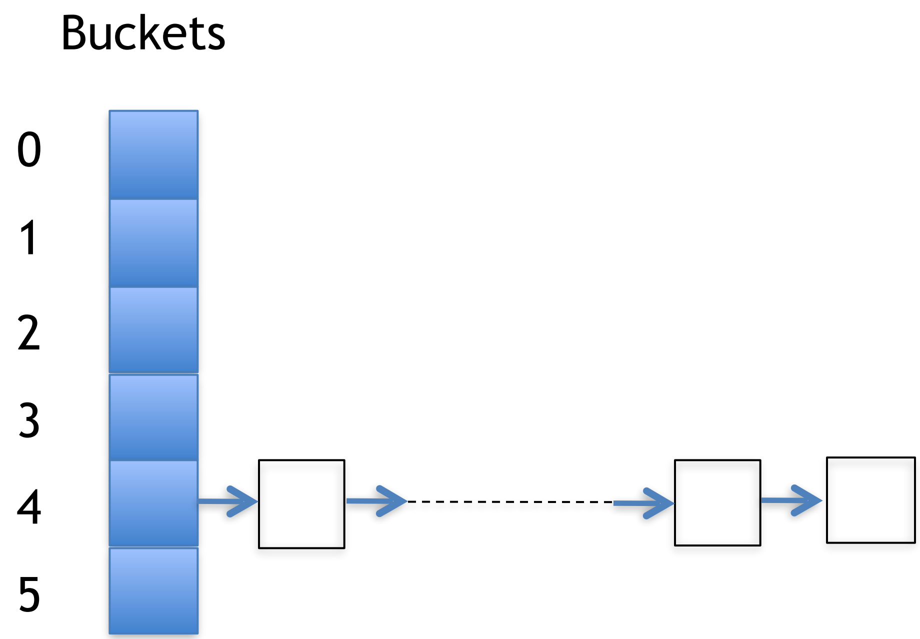
    $O(\log n) \rightarrow O(n)$
- Can the attacker supplies inputs to force the algorithm into the worst case expected running time?

Hash tables

Normal



Worst-case



Hash tables

- Reduce object to a (say) 32-bit hash value
- Identify the bucket to store the object
  - Hash value modulo the bucket count
- If two input objects map to the same bucket, a *collision* has occurred
- Buckets hold linked lists of objects (whose hash value module the bucket count is the same)
  - Hash chains

Collisions

Two reasons for collisions

- Hash values are identical
  - Find k_1, k_2, …, k_i inputs such that $Hash(k_1) = Hash(k_2) = … = Hash(k_i)$
  - Hash collisions
- Hash values are different but their values modulo the bucket count are identical
  - Find k_1, k_2, …, k_i inputs such that $f(k_1) = f(k_2) = … = f(k_i)$ where f is the function mapping inputs to a bucket (e.g., $f(x) = Hash(x) \pmod{n}$)

Attacking hash functions

- Different hash functions
  - Cryptographic hash functions: MD5, SHA-1, etc.
  - Non cryptographic hash functions: functions with 32 bits of internal state (speed!), e.g., XOR
- Attacking weak (non cryptographic) hash functions
  - Weakness: limited internal state
  - Idea: find input ("*generator*") such that internal state after hashing is the same as the initial state, say 0
  - Example: $k_1, k_2, \ldots, k_i$ such that
    $0 = Hash(k_1) = Hash(k_2) = \ldots = Hash(k_i)$
  - Then: any combination of $k_1, k_2, \ldots, k_i$ will be hashed to 0:
    $Hash(k_1k_2) = Hash(k_2k_1k_3k_2)$
- Finding generators: bruteforce

Attack on Perl hash function

- Crosby and Wallach on Perl 5.6.1 and Perl 5.8.0
- Hash function: state machine with 32 bit state
  - Input is mixed in one byte at a time, using addition, multiplication, shifting
- Finding generators: 46 generators in 1 hour time
- Inputs: ~90k strings of 24 characters
- Loading time
  - Malicious inputs: 6,506 seconds (1:48 hours)
  - Benign inputs: 2 seconds

Attack on Java hashCode function

```
uint32_t hash(const char *arKey,
              uint32_t nKeyLength)
{
  uint32_t h = 0;

  for (; nKeyLength > 0;
         nKeyLength -=1)
  {
    h= ((h << 5) - h) + *arKey++;
  }

  return h;
}
```

- Equivalent substrings
  - hash(‘Ey’) = 2260
    hash(‘FZ’) = 2260
  - hash(‘Eya’) = 70157
    hash(‘Fza’) = 70157
  - hash(abcEydef) = 2758600642447
    hash(abcFzdef) = 2758600642447
- Easy to generate infinite number of colliding strings (binary permutations)

Attacks

- Look at where hash tables are used:
  - IP addresses
  - Transaction IDs
  - Databases
  - URL parameters
  - HTTP headers
  - …
- See A. Klink and J. Walde, Effective Denial of service attacks against web application platforms, 28C3 (MOV)

Solving algorithmic complexity attacks on hash tables

- Better hashing functions
  - Cryptographic hash functions
  - Universal hash functions
    - Guarantee that given k_1 and k_2, the odds that $h(k_1) = h(k_2)$ are less than small value ε
  - Performance?
- Better data structures
  - Guaranteed runtime bounds, regardless of their inputs (e.g., red-black tree)
  - Implementation complexity?

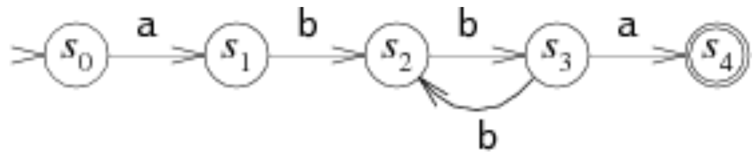
Regular expressions

- Notation for describing set of characters
  - Literal characters
  - Metacharacters (*, +, ?)
- abc*
  - Matches ab, abc, abcc
  - Does not match abcd

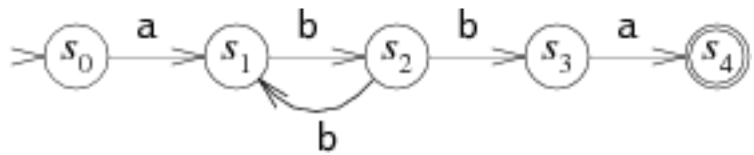
Regular expressions and automata

- Matching a(bb)+a

DFA



NFA



Examples courtesy of http://swtch.com/~rsc/regexp/regexp1.html

Matching regular expressions

- Backtracking
  - Whenever multiple paths are possible, take one (randomly) and record the choice
  - If matching fails, backtrack to last choice point (branch) and take alternative path
- The number of paths to check can be exponential in the length of the input string
- Attackers can create strings that force this worst-case backtracking
- Why using backtracking
  - Easy to implement back references

Attacks on regular expressions

- (a|aa)*b

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 1.0209710598

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 2.69697809219

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 7.27811908722

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 20.1901729107

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 50.6117949486

- [a-z]+@[a-z]+([a-z\.]+\.)+[a-z]+

redos@x................................ 0.192698001862

redos@x................................... 0.820160150528

redos@x...................................... 3.50762486458

redos@x....................................... 5.67420506477

redos@x.. 9.44989609718

redos@x... 15.4427139759

redos@x... 41.7495079041

Detecting vulnerable regex

Statically

- Understand why a regular expression causes blowup in running time
- Search for regular expressions that have the same characteristics
- James Kirrage, Asiri Rathnayake, Hayo Thielecke, Static Analysis for Regular Expression Denial-of-Service Attacks, 2013 http://arxiv.org/abs/1301.0849

Detecting vulnerable regex

Fuzzing:

- Generate inputs likely to cause issues with a given regex
- Measure the processing time
- Alert if it gets above a given threshold or increases faster than desired
- Microsoft, SDL Regex Fuzzer, http://www.microsoft.com/en-us/download/details.aspx?id=20095

Take-home points

- Denial of service attacks exploit imbalance between work done by attacker and work done by victim
- Computational complexity denial of service
  - Exploit algorithms that have bad worst-case running time…
  - By providing inputs that force the worst case to happen
- Examples: hash tables, regular expressions

Read more

- Solar Designer, Designing and Attacking Port Scan Detection Tools, Phrack 1998
  http://www.phrack.org/issues.html?issue=53&id=13#article
  - First mention of DoS attacks against hash tables
- S. Crosby and D. Wallach, Denial of Service via Algorithmic Complexity Attacks, USENIX Security 2003
  http://static.usenix.org/event/sec03/tech/full_papers/crosby/crosby_html/
  - First thorough discussion; found vulnerabilities in Perl, Bro (IDS), Squid
- A. Klink and J. Walde, Efficient Denial of Service Attacks on Web Application Platforms, 28C3 2011
  http://events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms.pdf
  - Break PHP, Python, Java, Ruby
- J. Aumasson et al., Hash-flooding DoS reloaded: attacks and defenses, APPSEC 2012

  https://131002.net/siphash/siphashdos_appsec12_slides.pdf

  http://emboss.github.com/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/
  - More sophisticated analysis of hash functions (differential cryptoanalysis)
- P. Junod, Like a Hot Knife Through Butter, 2012
  http://crypto.junod.info/2012/12/13/hash-dos-and-btrfs/
  - Attacking the btrfs filesystem

Read more

- OWASP, https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS
  - Examples of real-world, problematic regular expressions
- R. Smith et al., Backtracking Algorithmic Complexity Attacks Against a NIDS, ACSAC 2006 http://www.acsac.org/2006/papers/54.pdf
  - Attack against Snort intrusion detection system
- K. Namjoshi and G. Narlikar, Robust and Fast Pattern Matching For Intrusion Detection, INFOCOM 2010 http://ect.bell-labs.com/who/knamjoshi/papers/robustness-infocom10.pdf
  - More problems in Snort (back references)

Next time

- Sample exam