Memory Corruption Attacks

Previously Discussed
- Stack Smashing Attack
- Heap Overflows
- Format String Attacks

Integer Overflows

There are many ways in which an integer overflow attack can compromise the security of a system.

1. Assign a variable with more storage (say 16 bit - len) to a variable with less storage (say 8 bit – n).
   
   ```
   char buf[32];
   short len = read(fd, largebuf, sizeof(largebuf));
   char n = len;
   if (n < sizeof(buf))
       memcpy(buf, largebuf, len);
   ```

2. Assignment between variables of different signs.
   On assignment of variable which are handled as signed and unsigned under different scenarios integer overflows can occur.

   Consider the code fragment

   ```
   int i = len; //len is unsigned
   char out[256];
   if( i < sizeof(out))//Here this check succeeds because i is interpreted as signed
       memcpy(out,in,i); //memcpy interprets 3rd argument as unsigned
   ```

   If len is large then it is interpreted as a negative number when converted to a signed value. Then the check of i succeeds since it is negative, but since memcpy interprets it as unsigned, an overflow will occur.

3. Arithmetic Overflow
   Integer overflows occurring during the arithmetic operations.

   ```
   i = j + k;
   i = 4 * j;
   ```
These overflows are harder to control for + than for * because for + to succeed j or k needs to be close to the max value that i can store. For * a multiplication factor like 64 or 128 can cause the overflow to occur.

Example :-

```c
struct xx A[] = malloc(n * sizeof(struct xx));
```

n = value read by program from network.

Such code in programs may cause integer overflows. Similar code fragments are seen in programs used for manipulation of data using lists and arrays, Image programs, proprietary word processors etc.

Image programs may have a number stating the number of blocks of same size that will follow next, e.g. the frames of a animated image file.

```c
struct xx * buf;
//variable length array
int len = n * sizeof(struct xx);  //compute length
if(len < maxlen){                //check against max
    buf = malloc(len);        //allocate
    for(i=0; i<n; i++)
        readblock(buf[i]); //copy
}
```

Possible errors

n can be such that it has large value. So when it is multiplied by sizeof(xx), it can overflow the maximum value that can be stored and then become a small value.

If $n = 2^{28}$ and $\text{sizeof}(\text{xx}) = 16$, then $16 \times 2^{28} = 2^{32} = 0$

So if $n = 2^{28} + k$

Allocated data = $16 \times (2^{28} + k)$

So, if attacker controls n then it controls the amount of memory allocated precisely (which will be 16k in the above example.). Such attacks may cause writes to go way past than their limits.

Integer overflow attacks do not corrupt memory. They cause the intended effect of the bounds checks to be not achieved. Thus they take the program into a situation where the program with any kind of bound checks becomes vulnerable. Programmers are liable to miss the possibility of a overflow during a code review since they may look at the presences of bounds checks and conclude that there are no vulnerabilities.

**Memory Errors**

Several kinds of attacks against memory have been popular in the past years. Several defense mechanisms are also there for them. While new types of vulnerabilities have begun to emerge in the last few years, the most critical ones continue to be based on memory corruption.
Reasons for its popularity:

1. Memory errors are pervasive.

Most software today is written in C and C++, two languages that are notorious for memory errors.

Previously Java code was considered immune to memory errors, but now problems have been identified with the native code of JVM (Java Virtual Machine).

It was believed that Javascript operates at a higher level and should not be worried about to cause any kind of memory corruption. In reality Javascript has constructs which involve allocation and deallocation of large amount of memory. The Javascript interpreter that is built into most web browsers needs to do these allocations and deallocations. So, Javascript can be used for setting the stage for memory exploits. It can be used to download large code into the memory for the memory corruption attack to happen, i.e., preloading large-volume exploit code into web browser memory. In addition, if exploitation of a certain web browser memory vulnerability requires the heap to be arranged in a certain way, Javascript code can help get there since (a) Javascript code is typically written by the attackers themselves --- current browser/web architecture means that a browser executes the Javascript code provided by any web site it visits (b) Javascript code can allocate and deallocate objects, causing corresponding heap allocations/frees by the browser.

2. Present on large number of computers

Same memory exploit can be used against large number of computers. So memory exploits can be used for purposes like cybercrime which involve taking control of large number of computers for the purposes of spam and DDOS attacks. More client applications are present as compared to the server applications so more attacks are developed to exploit clients than servers. Web browsers were the main target in the early days of network-based attacks, but now the focus has shifted to client applications such as media players, word processors and image viewers etc.

If the data is complex then it is much more likely that the code for parsing this data has vulnerabilities. So attacks are developed which use documents like .doc or image files, which most users may not consider to be sources of exploits.

A memory error occurs when an object accessed using a pointer expression is different from the one intended.

It has 2 types:–

1. Spatial Errors – Out of bound access or Corrupted pointer
2. Temporal Errors – Dangling pointers
Defenses
There are two basic approaches:

1. Prevent exploits of memory errors
2. Prevent occurrence of memory errors.

(2) requires identification of all memory errors, and is often more difficult than option (1), so we focus on option (1) here, while providing a brief discussion of (2) here.

Two types of approaches can be used

1. (Static analysis) Scan code and tell the programmer about errors.

   False positive may cause programmer to look into more warnings. High false positive is bad but not too high a rate can be easily tolerated.

2. (Runtime blocking) When the error is detected at runtime, the victim program is typically shutdown --- we can't tolerate false positives here because any false positive causes the service to be hampered.

Runtime detection of Memory Errors
Most practical techniques today focus on spatial errors.

1. GCC has patch for bounds checking C, developed y Jones and Kelly. It was refined to reduce false positives in the CRED project. Its benefit is that it provides the highest level of compatibility with existing code, while its drawback is that it has high overhead (100% to even 1000% or more in some rare cases.)

2. Valgrind tool operates on binaries rather than source code.

   It can detect errors in heap allocated data. Information needed for detecting errors involving static buffer overflows and buffer overflows on stack is not present in binaries (you need info regarding variables and their sizes). Overhead is very large since Valgrind uses instruction emulation.

3. CCured detects temporal errors by not freeing memory.

   It uses garbage collection for freeing memory which works on the principle that a memory is freed only if there is no pointer to the memory location. The main benefit of this approach is its low overhead, but its drawback is that it may require nontrivial changes to existing code.

Block Exploit

1. Identify mechanism used for Corruption and block them.

   We have techniques which take a look at what data is being corrupted by the attack and then protect the data. E.g. Stack Guard, Magic Nos. and canaries in heap.

   We do not have general solutions for these but we use the way in which these happen to protect from the attack.

   The problem with this technique is that it protects only the targets and nothing else. E.g. Canaries only protect the return address. If there are attacks which do not affect the canaries then it cannot be detected by the mechanism as they are specific in detection.
2. Mechanism used for take-over
   Attackers control the behavior of victims program in some manner. There are certain actions that the victim program might perform which help the attacker to take control of victim. Example: action of using a return address to jump to or the action of overwriting the file name to execute. Our approach here is to disrupt what happens in the use of corrupted data. Randomization based defenses have been the most successful in this regard.

3. Mechanisms used for delivering payload
   For injected or existing payload we analyze on how to prevent payload to take over.
   NX – Non-Execute data segments
   CFI – Control Flow Integrity which ensures that the control flow transfers to legitimate locations and not the injected code.

**Issues for an attacker**

1. Attacker wants predictable results
   There is a difference between random errors and crafted inputs causing exploits. (This is also one of the main distinctions to be raised between fault-tolerance approaches and security.) For addressing random faults we can use probabilistic solutions but we cannot apply them in the case of intelligent adversary creating specific faults.

2. Using inputs under the control of the attacker
   If an attacker controls the program then he can put malicious code in the program, and no exploits are needed. However, the attacker typically does not have control over their target system, which is running benign but vulnerable software. The attacker wants to exploit vulnerabilities in such benign software by providing carefully crafted inputs.

Exploit software bugs that cause targets of writer’s to be controlled by inputs

1. Relative Address – Example: buffer overflow used to overwrite critical data. Attacker needs to figure out what is to be overwritten and know the distance where to write.
2. Absolute Address – Pointer corruption attack. Modify pointer values that are used to determine targets of jumps (or as locations of data in memory). Attacker needs to know the exact value of the pointer to be used for the attack to succeed.

In reality attacks are a combination of both.

Temporal Attacks have not been exploited so far. Spatial Attacks are widely used.

**Diversity Based Defenses**
Make sure that there is diversity among programs. Example: Memory errors are popular because they are everywhere. Now if there is an exploit in Internet Explorer then all the computers running it become vulnerable, and moreover, can be exploited with the exact same exploit. This enables
“mass-market” attacks where a single exploit is developed and can be used against all computers on the Internet.

We want to introduce diversity so that the same exploit cannot be used at such a large scale. We also want the diversity to be automatic and want it to be injected among the population of applications. We do not want the diversity to be arbitrary as it should preserve the functional behavior of the program.

**Automated Diversity Introduction**

This can be done on the basis of the semantics of the underlying programming language. This is done so that it can be applied to any program in that language. It will not break any programs. (It might break those programs which violate the semantics of the language.)

Use these techniques

1. Address Space Randomization
2. Data Space Randomization

These are complimentary in nature. ASR has been deployed now in OS’s and due to which some of the attacks if tried for experimental purposes on a system may not work. A workaround can be to understand the randomization and design the exploit accordingly.

1. Address Space Randomization
   In this we randomize the location of object in memory including code and data. Even if something gets overwritten by an exploit the chances of intended data being overwritten are less.
2. Data Space Randomization
   It randomizes low-level representations of data object. E.g. we can use 11110000 to represent numeric 0, rather than using 00000000. This makes it hard for an attacker to predict the correct meaning of data.
3. Instruction Set Randomization
   Randomize interpretation of low-level codes. It does not focus on the memory corruption, and can be thought of as a special kind of Data Space Randomization. Attacks protected by ISR are same as NX.
   Using these mechanisms, memory errors have unpredictable effects.

**Absolute Address Randomization**

We change the base address from where we start the memory allocation. We start from a random value and if needed, we wrap round the memory. So in this way a large amount of unpredictability can be introduced. Attacker may think that he is corrupting the correct data but the corrupted value, if it is a pointer, will end up pointing to the location of some random object in memory (or to unused memory). This technique does not help against attacks that involve no pointer corruption.
1st Gen ASR Techniques

It is developed 6-7 years ago. Randomizes the base addresses of all regions of memory. Data (stack, heap and static memory) and Code (libraries and executables) regions are randomized.

Processes each have their own private virtual memory space. As a result, every executable can assume that it will reside at a specific memory location, and hardcode references to these locations in its code. Such hard-coding mean that no address translations are needed at runtime or at load-time. But libraries can't make such assumptions since they need to co-exist with many executables.

Absolute address ASR has been adopted on UNIX and Windows Vista

Windows Vista – 8 bit randomization
   Win DLL's need to be aligned on 64kb boundaries so there is a 16 bit randomization limit on windows, which has been achieved on some implementations (but Vista designers chose to limit randomization to 8-bits.)
UNIX – 20 bit randomization is typical.
It is not more on unix because
   1. we need to fit various regions in memory without causing fragmentation
   2. libraries need to be aligned on page boundaries, which are typically 4K.

Stacks can have higher randomization than shared libraries which are page aligned.

There are 3 kinds of codes

1. Relocatable
   Used for DLLs on Windows. These use absolute address references in their code, so they must be loaded at a particular memory location. If that location is already occupied, then it needs to be “rebased” –– the references need to be updated to reflect the new location where it is loaded. Since there is no easy way to distinguish between address constants and data constants in binaries, DLLs need to explicitly list the locations that need to be updated during rebasing.

2. Position Independent
   Used for shared libraries in UNIX. This kind of library does not contain any absolute address-based memory references. Calls use relative distance between the caller and callee. For accessing data, a PC-relative addressive scheme is used. The content of the PC can be obtained by calling a routine, and having the routine move the return address to a register. The location of data objects can be obtained by using an offset from the PC. The advantage of this technique is that since it has no absolute references, it is possible to store the same library at different virtual addresses in different processes –– this avoids one of the problems with DLLs that need to be “rebased” and hence cannot use the same library image for two processes that need to load the library at different virtual memory locations.
Their drawback is that there is an additional overhead for PC-relative addressing, as described above. Nevertheless, on the balance, PIC has advantages ---- it does not require any “global” conventions across different software vendors in order to ensure efficient sharing of code image in memory.

3. Non-Relocatable
   Used for executables. Functions with specific memory addresses and they are used for their jumps. The advantage of this is that everything is known at compile time. All the symbolic references are resolved by the compiler, so no additional overheads are incurred at load-time or runtime.

   Executables become process with separate process memory so it causes no conflicts. However, since there is no way to distinguish constant values representing addresses from those representing data, there is no way to automatically “relocate” executables. As a result, the locations of code and data objects (specifically, static variables) used by the executable cannot be randomized. As a result attacks on static data, as well as return-to-exe attacks become possible. This possibility is exacerbated by some code sequences that may be commonly used --- For instance, Windows code often has Call (ESP) instructions that take the value of ESP as location of function and call it. By jumping to such an instruction, control is transferred to the top of the stack immediately, without requiring the attacker to know the address of the top of the stack. Since the stack top will typically contain attacker-injected code immediately after a stack-smash, this call (ESP) instruction can be defeat to defeat ASR even when just the exe code is not randomized.

Randomization defenses often have an all-or-nothing aspect to them. If we randomize 90% of the memory then the remaining 10% can be used for carrying out the attack. So we need to make it as close to 100% as possible.

**Limitations**

1. **Brute Force**
   Attacker can try all possible values for base addresses. They can try out 16 bit randomization options in 1 minute or so. The amount of protection is less because if the attack is against 100 million computers on the Internet then every 64,000th computer can be compromised by the attack.

2. **Relative Address Attack**

3. **Information Leakage Attack**