Attack types

Stack Smashing Attack

When a function g() is called by a function f(), first, the f’s BP is pointing at the beginning of the activation record of f(), when g() is called, first the return address is saved and then f’s BP must be saved in the stack, after that, g() gets executed.
An attacker can use a piece of malicious code to overflow the stack in order to overwrite the return address, so the return address will point to a piece of code that written by the attacker. When g() is finished, the program will continue to execute what the attacker has written.

- .Slide 2 of lecture slides shows simple program with a buffer and how it is represented on the stack, it also shows the stack grown downwards
    - .Address of buffer element calculated by : Starting address + 4 * X (e.g. the element is an integer)
        - .X is our element index
            - .Key point of attack is to cause a stack overflow which causes the return address to be overwritten and instead point to the buffer (See Diagram)
                - .Allows arbitrary code to be executed
            - .Other pointers can be attacked as well
                - .If F calls G
                    - .We now have a base pointer we can overwrite which would work the same as overwriting the return address (See Diagram)
- .Ways to Prevent
    - .Don't allow code execution everywhere
        - .Only allow at read areas or specify execute areas
            - .Microsoft and Intel have both added support for this approach to the problem
                - .Good idea but unfortunately attackers are clever...
                    - .If I can't inject my code, why don't I just use yours?
                        - .Nx cannot prevent the kind of attack in which an attacker will use existing code. An attacker can change the return address to a specific function, e.g. system function
                            - .Example: Attacker wants a shell
                                - .Return to the area in memory that invokes a shell
                                - .Function parameters are pushed onto the stack above the return address, so by overwriting these values, the attacker can change the parameters going to the system function!
                                - .These are often called return-to-libc attacks
        - .In Summary
            - .This causes attackers to be more constrained, but does not prevent all attacks
            - .Because of virtual memory the attacker can always predict the location/addresses of items and have found ways to deal with minor uncertainties
                - .For instance: Attacker knows address +/- 10 bytes
                    - .Insert nops to address this

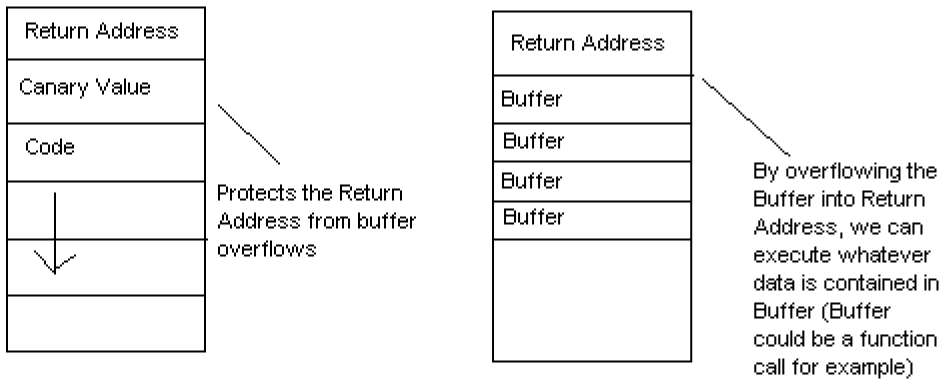
Guarding Techniques

- Stack Guard
    - One of the first
    - Put a “canary” value immediately before the return address(See Diagram)

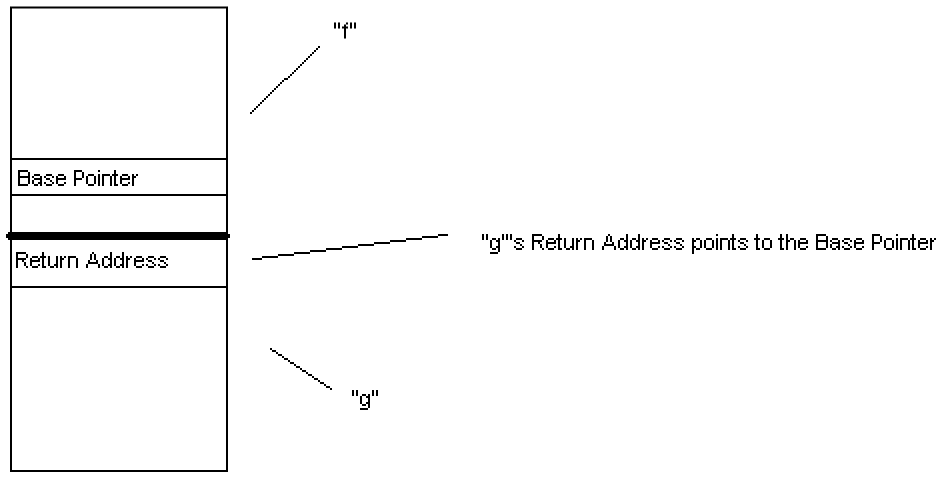
        - If value is overwritten we know an attack has happened then we simply abort the process
            - Different values can be used
                - Random value – otherwise an attacker can predict what the canary is and simply assign the same value when he/she overwrites the stack.
                - NULL - useful to prevent string copy attack.
                - Copy of Return Address— System will store a second copy of return address somewhere else, so if the return address in the stack is changed, we will discover that.
        - Major problem is this only protects the Return Address, other things such as a Base Pointer will be unprotected.
- Refinement of Stack Guard
    - ProPolice

      We can modify the canary value strategy such that we store simple local variables (rather than arrays) before return address and BP, then before simple local variables, we store canary value, so local variables can also be protected.
        - Change the stack around a little to confuse and mislead attackers

Reference Diagram

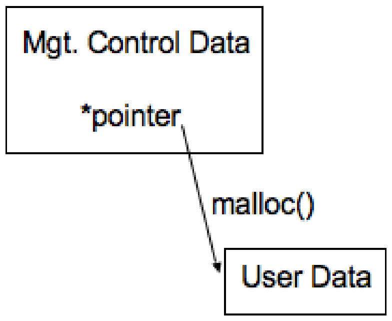


Return Address
Canary Value
Code
Protects the Return Address from buffer overflows
Return Address
Buffer
Buffer
Buffer
Buffer
By overflowing the Buffer into Return Address, we can execute whatever data is contained in Buffer (Buffer could be a function call for example)

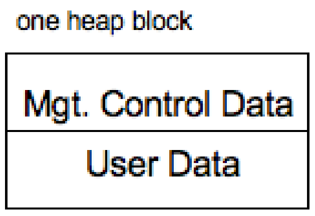


Heap Management

Before understanding how heap overflows work, is better to understand how heap management is done. One of the goals of heap management is to minimize the overhead of how information is stored. Heap management deals with a *free list,* of free blocks. One way of managing the free blocks is by keeping the Management Control Data separate from the User Data. When the user asks for memory, heap management allocates memory, gives it to the user, and keeps a pointer to this block of memory for bookkeeping purposes.



Another way is to have them both, Management Control Data and User Data, side by side in one data structure. Control Data preceding the User Data. In this way, one could imagine that heap management uses a double linked list to keep track of the free heap blocks.

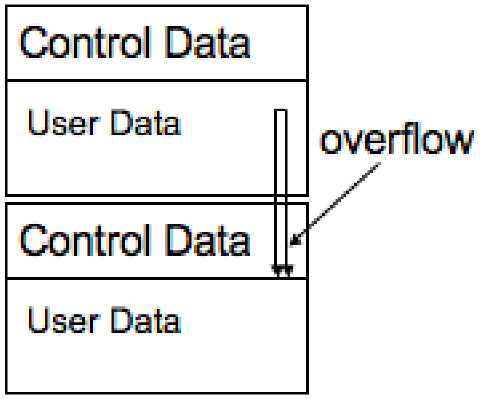


Keeping separate data structures complicates things. Allocation of data control structures is separate from allocation of user data structures; heap management has to keep track of more storage units. By putting the Control data and user data together in one block, one avoids this complexity, and reduces the level of indirection. This might help to the improvement in performance. For the rest of these notes, we will only consider the heap management implementation where control data and user data are kept in the same data structure. Most typical heap management systems implement this way of management too.

Heap Overflows

Let us say there is a heap block A, followed by another heap block B. When writing to a heap buffer in A, a heap overflow is possible if this buffer extends past the end of A and reaches into B. If it does, this buffer will overwrite control data in B.

There are many ways in which this type of overflow can be exploited, but they all are similar. An example of how this vulnerability can be exploited is by overwriting two fields in the control data section of an adjacent block. These two fields are next



and previous (remember free blocks are kept in a doubled linked list). The part of the heap management that adds and removes free blocks trusts the values referenced by next and previous, and this provides attackers with a way to mount an attack.

Suppose that an overflow occurs as shown in the above figure. Let us say that the overwritten control data belongs to a block in the freelist that is then allocated (as a result of a malloc that follows the copy operation that led to the overflow). To delete this block foo from the freelist, the heap management code might execute two statements:

foo->prev->next = foo->next;

foo->next->prev = foo->prev;

Focusing on the first statement, the effect of this statement under normal conditions (when the prev and next pointer fields stored in foo are valid) is shown in the figures below, where foo is the middle block in the chain.

Figure A

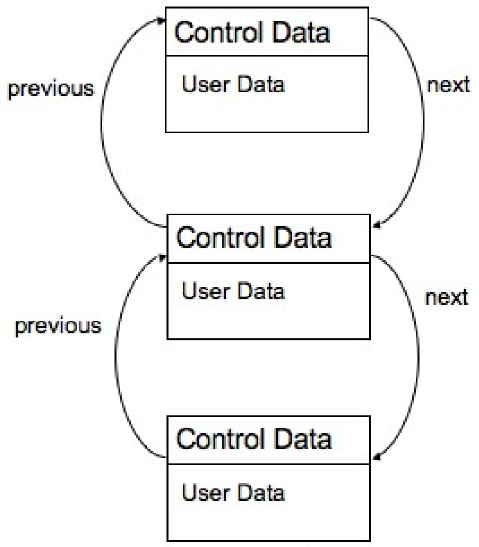
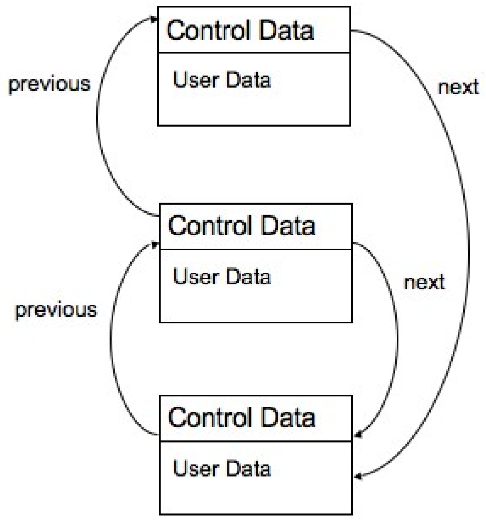


Figure B



However, if the attacker controls the values of these pointers (as a result of the overflow mentioned above) then the effect is quite different. In particular, assume that the attacker has overwritten the “prev” field with a value “a” and the next field with the value “b.” Then the effect of the italicized assignment above, when executed by the heap management code, is:

a->next = b,

Note that effect of the above statement is to store the value “b” at the address “a,” if

“next” is the first field in the structure defining the heap blocks. (If it is not the first field, then “0” will be replaced by a different constant value that represents the offset of the “next” field from the base of the structure.)

In effect, the heap management statement above allows the attacker to write an arbitrary word (“b”) at an arbitrary memory location (“a”). This can be used to carry out attacks. We point out that this corruption need not be triggered just by a malloc operation. It is possible to trigger similar attacks when a block is freed, or is merged with a preceding or succeeding blocks, although the exact details of the attack will differ.

Some of the possible targets that can be overwritten with a heap overflow attack:

1. **Function pointers** (code pointers) are the most attractive to attackers. A corrupted function pointer can point to code provided by the attacker and also already existing code.
    - **1.a.** **Return Address** in a stack. In this case, the specific location of a return address is needed, and so, the canary mechanism used to detect a stack smashing attack will not work in this case. However, a second copy of the RA (return address) will continue to help.
    - **1.b.** **Global Offset Table:** (dynamic linking) Table of function pointers in writable storage because it has to be filled in at run time. A two-step process is done to fill in this table. The program will not know where the library is, the library location will be known at run time, when the library is loaded. A table is constructed with n slots, where n is the number of functions used by the program. Say the program calls function f(), and the third slot in the table is assigned to f(). So the call of f() will be replace with call *func_table[3]. At run time, the dynamic linker fills in the table. Attackers want to execute system calls, so an attacker can overwrite a GOT entry corresponding to commonly called functions such as “read” so that it points to the attacker's code.
    - **1.c.** **Function Pointers in static memory.**

2. **Data Pointers**
    - **a.** Name of programs executed or files opened.
    - **b.** Application specific data, for example: a login program uses different methods of authenticating a user within a loop, and when one of these methods succeeds, the program may set a variable named “is_authenticated.” When this variable becomes true, the program may break out of this loop and proceed to execute a shell. An attack can change the value of this flag so the program breaks out of the loop and logs in the attacker.

Defenses against heap overflow attacks
Heap canaries: put a canary between two heap blocks. Depending upon the implementation details, the canary at the end of each heap block, at the beginning of blocks, or in the middle of the control data. This is what most typical systems do: Linux, Windows. This kind of protection is not very expensive.

In general, separating user data from control data is a good idea, it makes the program less vulnerable and harder to exploit, and avoid the idea of an attacker changing the flow of control of a program, which can be more powerful than an attack that just changes the data. We mentioned the example of separating heap control data from heap user data, but the same high level concept can be thought as being operational in the context of stack-canaries, or ProPolice.

Between data attacks and injected code attacks, attackers might choose injected code, but attacks on data can also be very powerful, as described in the previous example of changing the login program.

Format String Attacks
printf, printing to a file buffer
sprintf, write to a character array

These functions take variable number of arguments. The callee does not know how many parameters were passed. The callee has to figure out how many arguments were passed by looking at the format argument, and read them from the stack. If an attacker controls the format argument, she can fool the printf routine to read an arbitrary number of arguments from the stack. Moreover, since the format string controls the operation of the printf-family of functions, this control basically allows the attacker to exert significant control over the code executed by the victim program.

A format string vulnerability exists in programs that read a string from an untrusted source (e.g., a socket connection with a remote site) into a variable *s* and uses a statement

        printf(*s*)

to print it, instead of the more secure form

        printf("%s", s)

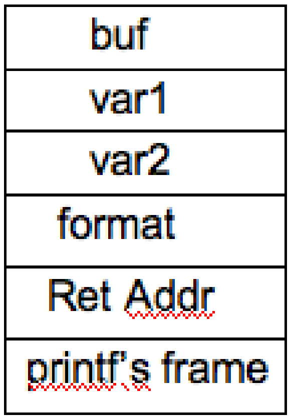
Now, by providing the value of "s, the attacker can control the effect of printf.
, or by making it think that a different number of parameters than the actual number of parameters are being passed. Still, not all problems are solved. In particular, the attacker wants to write something into memory (e.g., overwrite return address) in order to gain control, but printf only reads from memory. Well, almost. There is an obscure "%n" format directive that involves writing to a specified memory location. But the data written is not directly controllable: it is the number of characters successfully printed so far. But the attacker can control this value by setting the field widths while printing. Moreover, by using a "hh" prefix to "n," the attacker can write just the least significant byte of the number of characters printed. For example, the attacker can write "50" into location x and a "30" into location x+1 using the following format string, provided that he can arrange the %n parameters to reference the values "x" and "x+1" that must be on the stack.
"$\underline{\text{\%50d \%hhn \%206d}}$ %30d %hhn"
$\quad\quad\downarrow$

Note that after %206d is done, printf would have printed 256 characters, i.e., its least significant byte will be zero. So, when %30d is carried out, the next %hhn will print 0+30 = 30.

How does the attacker ensure that "x" and "x+1" appear on the stack. Usually, format string vulnerabilities are exploitable when the format string argument *s* described above resides on the stack. For example, consider the following vulnerable function:

```
void f() {
    char buf[256]
    int var1
    int var2
    read(d, buf, 256)
    printf(buf)
  ...
```

printf interprets whatever is on the stack as parameters. When printf is called we look at the stack. The "format" variable is the first parameter taken by the printf function. When f calls prints, it would have stored the location of *buf* in that location. Assuming that the compiler allocates variables in the order in which they were declared, we might see the local variables of the caller (i.e., function f) above this parameter.
The way parameter passing works, printf will interpret the words in the stack as parameters. Thus, the location corresponding to var2 will be interpreted as the second argument, var1 as 3rd argument and so on.



Note that the attacker can start referencing *buf* as the 4th parameter – from this point on, the attacker controls the addresses corresponding to the %n argument.

Format string vulnerabilities are very specific and can be easily avoided. Just look for printf calls where there is just one argument to printf, and make a quick check whether the data being read is a static string, or is it being read from somewhere else. A more general defense would ensure that the variable argument feature is being used securely, e.g., the callier can specifically send in another argument that indicates the number of parameters.

Integer Overflow

There're multiple forms of integer overflows:
Assignments between variables of different width. E.g. assign a 32-bit value to a 16-bit variable. In this case, the higher 16-bit will be discarded.

Assign an unsigned integer to a signed integer variable. If the unsigned value has "1" on the highest bit, after the assignment, the signed variable will be an negative integer.
An integer overflow can cause heap overflow if you allocate less memory space than needed.

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, nut 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 :-

```
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.

```
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 sizeof(xx) = 16, then $16 * 2^{28} = 2^{32} = 0$

So if $n = 2^{28} + k$
Allocated data $= 16 * (2^{28} + k)$
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:

.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.

.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:-

2. Spatial Errors – Out of bound access or Corrupted pointer
3. 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

- we need to fit various regions in memory without causing fragmentation
- 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 prticular 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

Limitations of Absolute Address Randomization

It’s not possible to achieve complete randomness in ASR. Some memory regions, such as libraries, may be required to be aligned on page boundaries, thus making their lower-order bits predictable. The attacker can corrupt these bits in a predictable way.

Brute force attacks are still possible in which the attacker keeps trying different values till she succeeds. A successful attack was carried out several years ago by Shacham et al on 16 bit randomness of ASR implementation (called ASLR) by the PaX project. The attack succeeded by trying 32K values in around 2 minutes. This attack shows that an E-commerce server cannot rely solely on ASR as a defense. At the minimum, contingency measures need to be taken when repeated crashes are observed.

Also note that if you consider an Internet-wide worm, ASR offers only limited protection. In this case, the worm may try to attack millions of computers on the Internet at the same time. With 16-bit randomization, the attack will succeed on 1 of 64K computers on the very first attempt. Thus, in this scenario, even a contingency measure based on repeated crashes does not help. It is likely that the spread of the worm will be slowed down, as it is going to take many more attempts to infiltrate into a system due to the use of ASR.

Partial Pointer Overwrite: The concept of PPO is to modify the least significant bytes. In x86 it is possible to change least significant 1-2 bytes of a pointer. If the randomness is contained in the upper 16 bits of a pointer, this means that the attacker can predict the lower 16 bits. For instance, it is easy to mount return-to-libc attacks using this approach. The RA on the stack is going to correspond the the location of some instruction in a library (or the executable). Since the relative distance between this instruction and other code in the library, he can simply modify the least significant 2-bytes of the RA so that it will now point to the code of attackers choice that resides within the same library.

Note that the attacker has to rely on other vulnerabilities than strcpy for PPOs since a strcpy will write a null-byte following attacker-provided bytes, which will corrupt the next byte in the pointer.

Many integer overflow vulnerabilities result in an out-of-bounds access, which involves a base address and an offset. It is the offset that is involved in overflow, while the base address remains fixed in the program. So, in this case, we essentially have a relative address attack, which AAR does not protect against. However, sometimes the offsets can be very large, resulting in an address that goes past the end of a memory region (like static memory) into another region (like the heap). In this case, AAR can help since the distance between two regions becomes random with AAR.

Information Leakage Attacks: An address value escapes and gets back to the attacker, e.g. a server can send back some data to the client, who is actually an attacker. Due to a bug in the system, the server may send more data than is intended, for instance, it may malloc a buffer of 64-bytes, fill it with 32-bytes of data, but then send all 64- bytes back. The uninitialized 32 bytes may contain pointer values. By comparing these values with the same values on an instance of the victim program on attacker's own machine, she can identify the random offset used by the AAR scheme. After this she can correctly guess all the addresses of all data and data locations at the server.

ASR is still not very widely deployed. As a result the feasibility and ease of information leakage attack is not well known. It may very well turn out that information leakage vulnerabilities are common – if so, AAR will not provide any protection at all.

One fundamental deficiency of AAR is that the mechanism relies on a single random number. When the number is leaked, the entire system becomes compromised. A proposal can be to employ various random bases so that even some of the bases are compromised; the rest of the system stays trusted.

More generally, AAR is not effective against data attacks or other attacks described above that rely only on relative distances. This motivates the relative address randomization techniques described below.

Relative address randomization

An improvement over the Absolute Address Space Randomization is the Relative Address Space Randomization or RAR. In RAR, the relative addresses of all individual code and data objects are randomized. RAR focuses mainly on static and stack variables, and code. It does not focus on heap since relative address randomization in heap is somewhat easier to achieve --- simply change the malloc library.

The relative distance between different functions is also randomized. Code transformation can be used to permute the relative order of routines in memory. Changing the relative address of existing functions thwarts return-to-libc attacks.

An improvement to this kind of randomization is achieved using write-protected memory pages between two pages to prevent overflow. If an overflow occurs in one of the segments that flow beyond its boundary, it will go into the protected page and a memory fault will arise. Implementing this improvement entails space overhead. Another price is that each page that needs to be protected requires a system call. If you have to introduce 100K pages that are protected, then it will entail significant startup overhead.

A compiler can allocate two static variables in any order as the C language doesn’t place any restrictions or norms about this. As a result an overflow in one variable cascades to the other. With RAR, each time the program is run, the relative positions of the variables are different so it can never be predicted if the overflow in a particular variable will reach to some other particular variable, which is the basic requirement behind most attacks.

Side discussion: the Propolice defense changes the relative order of some local variable, but this is done in a specific way that maximizes the effectiveness of the canary. In particular, there is absolutely no randomness involved, and the relative distances can be predicted.

Compile time randomization is not very useful. This is because, with current distribution models, the same binary is distributed and installed on all machines. Thus, a compile-time randomization won't contribute to any diversity across the population. Even if the installation model changes so that each software download is a differently compiled binary, there is still the problem that the resulting binary would have the exact same randomization each time it is run. This will mean that attackers can monotonically gain information with each attack attempt. In contrast, if the randomization is determined at load time or runtime, each execution is different, and an attack during one run will not yield information regarding randomization that is useful for another run.

With all the security mechanism in place, some random attacks may still succeed but it’s not possible for an attacker to launch a predictable attack (unless the attack is repeated many, many times.)

For security against stack-smashing attacks, there is another mechanism that can be used: split the stack into into two parts. One part contains the values that are usual targets for overflows, which are mainly control data like Base Pointer and return address while the other part contains variables that are prone to overflow, e.g., arrays. The two stacks can be back to back, possibly with a protected memory page inbetween to prevent any possibility of overflow from one stack to another. Since the first stack does not contain any object that can be involved in an overflow, the values on that stack are safe from being corrupted by a buffer overflow. (The RAR technique described below uses this dual-stack approach, and randomizes the order of variables on the second stack, while performing no randomization on the first stack.)

Benefits of RAR:

Higher entropy: The attacker needs much more information to break the RAR as the randomization is done at a finer granularity. Up to 28 bits of address can be randomized, as against 16 bits of AAR.

Information leakage attacks are not effective --- even if some information is leaked regarding the location of one object, this does not help the attacker know the location of other objects in memory.

For heap overflow attacks, the attacker needs to guess 2 pointer values: first, the location of a function pointer that he wishes to change, and second, the location of the code that this function pointer should be pointed to. With basic AAR, it is typically the case that once the first pointer value is correctly guessed, the second one can be guessed as well, since every thing is based on a single random value. With RAR, this is no longer the case.

Runtime randomization of each variable requires extra information as the binaries have no information about variable boundaries. In the RAR technique described here, additional information is encoded into the binary that helps achieve relative randomization. The details can be found in this paper.

DATA SPACE RANDOMIZATION:

The data space randomization focuses on the randomization of interpretation of data. The main concept behind DSR is to randomize the effect of overflows so that the effects of data corruption become non-deterministic.

In the context of DSR, it is important to realize that data internal to a program can be represented

internal to the program in any way that the program chooses. For data that is involved in external communication, this is not true.
(See the slides used in lectures for more information. You can also look at this paper for a more detailed presentation.)
One thing to notice is that for all randomization problems, we essentially consider weak adversaries, i.e. those who cannot run a program on the system. If an adversary can already run code, then randomization does not provide much protection, as the running code can scan the memory to identify the random values being used.

Benefits of DSR:

Provides greater entropy as it makes 32 bit randomization possible. It can use different masks for different variables so each overflow can be immediately detected at the time the data is examined. It protects all data, not just pointers, and it is effective against relative address attack as well as absolute address attack. It can detect intra-structure overflows which are difficult to detect using any other mechanism.

Intra-structure overflow:

In a structure, each variable is saved at a location that is a multiple of its size and the leftover space is left blank.

```
Struct   {
         char a;
         short b;
         double f;
         int d;
 }
```

Base +0	a
	Empty space
Base +2	b
	Empty space
Base +8	f
Base +16	d

A processor architecture (and possibly, a compiler) defines certain aspects such as sizes of primitive data types such as short and int are defined, as are their alignment requirements. For instance, on the x86 architecture, integers are 32-bit while shorts are 16-bits. In addition, integer variables are located at a multiple of 4-byte (i.e., their address has two zeroes at its LS bits). As a result, when the fields in a structure are laid out, some gaps may need to be introduced to satisfy the alignment requirement. These distances remain fixed, and must remain fixed, according to the C language semantics. In particular, if you change the size of gaps or introduce additional padding where it is not required, this will break working programs. As a result, it is not possible to use relative address randomization to defeat buffer overflows from one field of a struct to another. Indeed, many of the techniques for complete memory error protection do not offer any protection against such field-to-field overflows. With DSR, it is possible to simply use a different mask for each field, thus ensuring that such overflows become unexploitable. (To do this, it must be the case that the alias analysis report that the two fields in question aren't involved in aliasing --- this will not necessarily hold in all programs.)
Aside: Pointguard was a mechanism that was proposed earlier to protect against pointer corruption attacks. It relied on xor'ing pointer values with a bit-m ask. The problem with the technique was that it did not consider aliasing and hence would break some legitimate programs.

Special precaution is required to deal with aliasing. Two pointers pointing to the same data should not have different representations: otherwise, different masks may be associated with the access to the same data just because the pointer variables are different. For instance, consider:

```
struct XX a  =  malloc (sizeof (struct XX));
bzero(a,sizeof(struct XX));
```

where bzero is the standard C-library function used to zero out a block of memory. It is declared as follows:

```
Void bzero(char *b, int n);
```

As the function bzero will write ‘0’ to all its fields while the fields may have a different representation for ‘0’. To ensure that this does not happen, the DSR technique uses a source code analysis to identify aliases, and ensures that the mask associated with a memory location stays the same, regardless of how this memory location is accessed.

Implementation:
Use source to source transformation. To achieve better performance, the technique is applied to buffers and pointers only. To minimize the likelihood of attacks due to the aliasing limitation mentioned above, it modifies the memory layout so that adjacent buffers use a different mask.

AAR doesn’t cause any serious interoperability issues, RAR causes some (or at least suffers the problem of needing source code access.) But DSR introduces interoperability issues with existing libraries. To make it work correctly, all the libraries have to be analyzed and transformed. (To a varying degree, this is a common problem with all the memory protection mechanisms.)

Bounds – Checking C [Jones and Kelly]

Every time memory is allocated, the beginning and end of the variable (the range of the object's address) are stored in a data structure.

4. For efficient access, a Splay Tree is used
5. Pointer arithmetic and dereference operations involve a search, so speed is important to minimize overhead

This is done for each static variable at the beginning of the program.
At the entry of each function, memory regions corresponding to each of its local variables need to be inserted into splay tree; just before return, these regions have to be deleted from the tree. Finally, the region returned by each malloc operation needs to be entered into the tree, and the free operation should delete that region from the tree.

Whenever a pointer is dereferenced, the pointer value is checked in the splay tree to see if it corresponds to a valid object. If not a memory error is flagged and the program aborted.

Just checking pointer dereferences is not enough: it wont capture out-of-bounds access, since the access would still correspond to a valid object, but not the intended object. (Typically, an out-of-bounds array access ends up accessing an adjacent object in memory.) At the point of dereferencing, there is simply no information to help determine if the target being accessed is the intended object: we have just a single pointer value, which has already advanced beyond the intended object. So, the only way to check if a pointer is going out-of-bounds is to "catch it in action." In particular, pointer arithmetic operations need to be checked to determine if they cross an object boundary, and if so, an error should be flagged.

Consider a pointer-arithmetic operation: q + e, where q is a pointer variable and e is an integer expression. To prevent out-of-bounds access, we need to check that q as well q+e fall within the memory region corresponding to a single object. Otherwise an error is flagged.

The problem with this approach is that it performs "eager checking" of pointer values. In particular, it seems reasonable to perform arbitrary pointer arithmetic if the result is never dereferenced. So, a better alternative is to simply set the pointer value to an invalid value rather than immediately flagging an error when the pointer arithmetic leads to an out-of-bounds pointer. Unfortunately, even this does not always work with all programs. Consider the following loop that initializes all elements of an array with zero, except the last element which is set to 1.

```
int a[n];
for (p = a; p < &a[n]; p++)
        *p = 0;
p--; *p = 1;
```

Note that when the loop is exited, p goes outside the range of array a. Thus, the last pointer arithmetic operation p++ will lead to an out-of-bounds pointer. If p is reset to an invalid value (e.g., NULL), subsequent operation *p = 1 will lead to a runtime error. However, there is no memory error in this program.

To avoid raising errors in correct programs, Bounds-Checking C extends arrays by 1 element. Unfortunately, this can still lead to problems in the following variant of the same loop:

```
int a[n];
for (p = a; p < &a[n]; p += 4)
        *p = 0;
p -= 4; *p = 1;
```

In this case, every 4th element is initialized to zero. In this case, the last pointer arithmetic may take the pointer value to as many as 3 elements beyond the end of the array a. Since the padding is only one element, Bounds-checking C will result in a runtime error on the above program.

CRED [Ruwase et al]

To avoid the problem experienced by bounds-checking C, a simple technique was proposed by Ruwase et al and was implemented into a system called CRED. If a pointer goes out-of-bounds, it is set to point to a special out-of-bounds object that is stored elsewhere in memory, for instance in high addresses. These special out-of-bounds objects are designed to store information about the original pointer, most importantly the base address, the bound, and the value of the pointer when it was changed to the out-of-bounds object. The purpose of these special objects is to store information about a pointer so that it can be later restored, for instance when it returns back within the bounds of the original object.

For instance, an object starts at address 100 and ends at 200. A pointer p points to this object. The program temporarily assigns p to 204, meaning that it cannot be de-referenced (since it would be out-of-bounds of the object). So, a metadata object is created to store information about this pointer --- the metadata includes the base, bound and the current value of the pointer.
This metadata object is allocated in high memory --- for instance, on Linux, the maximum usable memory address is typically 3GB. The compiler can allocate a region next to the highest addressable memory to store these metdata objects, say, from 3GB downto 3GB-100MB. Each time a pointer goes out of bounds, a metadata object can be allocated in this memory region, and then the out-of-bounds pointer value is changed to point to its metadata.

The purpose of using a higher value for out-of-bounds pointer than any valid pointer is so that such out-of-bounds pointers can be easily recognized and processed in a special way. This special processing can be understood best in terms of a source-to-source transformation. Consider the following code:

```
p = p – 4;
*p = 5;
```

This gets transformed as follows:

```
if (p > max_valid_address) /* say, 3GB-100MB in the above example */ {
   p->value = p->value – 4;
   if ((p->value >= p->base) && (p->value <= p->bound)) {
      tp = p;
      p = p->value;
      free_oob_object(tp);
   }
}
```

```
else if (goes_out_of_bounds(p, p-4))
    p = allocate_new_oob_object(lookup_base(p), lookup_bound(p), p-4);

if (is_valid(p)) /* look up splay tree to determine if p's value falls within
                        the range of some valid object*/
        *p = 5;
else .... /* flag a runtime error */
```

Benefits and Drawbacks of Bounds-Checking approaches

The bounds-checking approaches described above are highly backward-compatible: if malloc/free calls are wrapped so that they insert/delete from the splay tree, then instrumented programs can work with uninstrumented libraries in most cases. In particular, in most such scenarios, the executable , which we are assuming to be instrumented, will end up calling the library. If the library returns an object, that will have been allocated typically on the heap. Since we wrapped malloc/free, information about such objects would still be in the splay tree. This is unlike other memory error detection techniques that typically require all libraries to be instrumented.
Note that the above benefit does not work in cases where an uninstrumented library passes a statically or stack-allocated object to instrumented code. (This is unusual.)

The drawback of bounds-checking approaches are as follows:
(a) pointer to integer casts may not work as expected. In particular, such casts may result in memory errors going undetected.
(b) pointer-to-pointer casts may implicitly result in pointer-to-integer or pointer-to-nonpointer casts if the the pointer objects are structs that in turn contain a combination of pointer and non-pointer fields.
(c) unions of pointer and non-pointer types lead to the same problem as (a), while unions consisting of different structure types lead to same problems as in (b)
(d) object-to-object copies may lead to a similar problem if the source and destination objects have different type, and contain a combination of pointer and nonpointer values.
(e) temporal errors that occur due to reallocated memory are not detected.

Example of case (b):

```
struct A{
        int i;
} a;
struct B {
        char *p;
} *b;
b = (struct B*)&a;
```

This code is permitted in C language, but causes the integer field value i to be interpreted as a pointer field value p. If p is subsequently dereferenced, the best we can do is to check if p corresponds to a validly allocated object; we have no way to check if p was fabricated, or if it resulted from invalid pointer arithmetic operations.

<u>Detecting all memory errors at runtime</u>

Smart pointers – store base and bound values
- .stored alongside data
- .problem: messes up the layout of data structures
- .as a result cannot link code with smart and regular pointers due to inconsistency

> Compatibility is a major issue. E.g. a pointer with "in-band" metadata will be 20 bytes, while ordinary pointer has only 4 bytes. When you try to combine two pieces of codes with one of them having metadata and the other having not, problems arise: in particular, a structure containing pointers cannot be passed by instrumented code to an uninstrumented library since the two will have different sizes.

An alternative approach is to store metadata "out-of-band" --- that is, we store them separately. For instance, for each pointer variable p, we can create a variable p_info to store its metadata.