ECE 650 - Midterm Exam Sample Questions

Below find the mid-term exam instructions.

- Please read all instructions carefully.
- There are 3 questions on this exam, all with multiple parts. This exam will be graded out of a maximum of 100 points.
- Please submit all solutions via the appropriate dropbox on LEARN. No other form of submission shall be accepted. We prefer that you submit your solutions exactly once.
- You have 2 hours and 30 minutes to complete the exam. If you are not able to submit within the time allotted, you will get 0 points.
- The exam is open book and notes. You are allowed to consult resources on the internet. However, you are not allowed to simply copy solutions from any source. Instead, your solutions must be in your own terms. Further, you may not consult with each other or any other person, other than the professor and/or TAs. You can reach us via Piazza or email.
- **For each question below, please write your answers in the space provided right after the question.** Please make sure to write your names and IDs on the first page of this exam book.
- **You may submit your solutions as a PDF file, generated using MS-Word or Latex editor or some other suitable editor. We will provide a editable version of the questions. Hand-written solutions shall not be accepted.**
- Solutions will be graded on correctness, clarity, completeness and brevity. Most problems have a relatively simple and straightforward solution. We may choose to give some credit for partial solutions.

NAME: ____

Email ID: ____

Student ID: ____

In accordance with the letter and the spirit of the University of Waterloo honor code, I pledge that I will neither give nor receive assistance on this examination.

SIGNATURE: ____

Problem	Max points	Points Received
Q1	30	
Q2	40	
Q3	30	
Total	100	

Question 1: Regular Expressions (30 points)

(Note that this is a list of 12 sample sub-questions and are provided for you to deepen your understanding of the concepts in class. The actual exam will have 3-4 sub-questions for each question.)

Each of the questions below are worth x points. Please provide a pithy, precise, correct and complete answer. You only need basic regular expression (regex) operations to answer these questions.

1. **Language to Regex:** Write a regular expression for the following language L presented in set theoretic notation. Justify your answer informally:

$$L : \{w \mid w \text{ starts and ends in the same character from the alphabet}\}$$

Solution: The regex R below characterizes L exactly.

$$R : 0(0+1)^*0 + 1(0+1)^*1 + 0 + 1$$

To formally establish the equivalence between the language of any regex R and any other given language L, we have show that every string generated by R is in L and every string in L is in R.

(a) Every string generated by R above is in L: This can be seen by observing that every string generated by R starts and ends with the same character.

(b) Every string in L is generated by R: To see this, observe that every string in L that starts with a 0 and ends with a 0 is characterized by the sub-regex $0(0+1)^*0 + 0$. The proof for strings starting and ending with a 1 is symmetric.

Note that this language L does not contain the ϵ (empty) string since the language specification requires at least one character must be present in every string. Hence, ϵ is not in R.

2. **Equivalence of Regexes:** Is the following sentence S true? If your answer is YES, provide a justification. Else, provide a counterexample.

S: The following two regular expressions represent the same language over the alphabet $\{0, 1\}$:

$$r_1 = 00(0+1)^*00 + 00(0+1)^*11 + 11(0+1)^*00 + 11(0+1)^*11$$
$$r_2 = (00+11)(0+1)^*(00+11)$$

Solution: True, both regular expression accepts the language for strings start with 00 or 11 and end with 00 or 11.

3. **Equivalence of Regexes:** Is the following sentence S true? If your answer is YES, provide a justification. Else, provide a counterexample.

   S: The following two regular expressions represent the same language over the alphabet $\{a, b\}$:

$$r_1 = (ab^*)^* + (a^*b)^*$$

$$r_2 = (a + b)^*$$

   Solution: False.

   Consider the string 'baba'. This string is clearly part of the language characterized by the regular expression $(a + b)^*$ (the universal language over the alphabet $\{a, b\}$, however is not part of the language described by the regex r_1. The reason is that, any string accepted by r_1 must start with an 'a' or end with a 'b'. Unfortunately, the string 'baba' does not satisfy either of these properties.

4. **Language to Regex:** Let $\Sigma = \{0, 1\}$, construct a regular expression R for the following language L. No need to justify the correctness of your answer.

   $L = \{w \mid w$ starts with 1 and ends with 0, and w has odd length$\}$.

   Solution: $R = 1(1 + 0)((1 + 0)(1 + 0))^*0$.

5. **Language to Regex:** Let $\Sigma = \{0, 1\}$, and let L be the language of the set of all strings containing sub-string 00 exactly once. Construct a regular expression that accepts L. Justification not required.

   Solution: $R = (1 + 01)^*00(1 + 10)^*$

6. **Language to Regex:** Give a regular expression for the following regular language, assuming the alphabet is $\Sigma = \{0, 1\}$:

   The set of all strings containing at least one 1, and at most one 0. Justification not required.

   Solution: $1^+ + 1^+01^* + 1^*01^+$

7. **Regex to Language:** Describe the set of strings the following regular expression accepts, in English:

$$1^*0(01^*0 + 1)^*$$

   Solution: Strings containing an odd number of 0s.

8. **Language to Regex:** Consider the language of all binary numbers (i.e. strings over the characters '0' and '1') which are congruent to 3 modulo 4. Give a regular expression for this language. (Hint: numbers which are congruent to 3 modulo 4 have a remainder of 3 when divided by 4.)

   Solution: The regular expression $(0+1)^*11$ describes the language.

9. **Language to Regex:** Let $\Sigma = \{0, 1, 2\}$, and let L be the language of the set of all strings containing odd number of 0's. Give a regular expression that generates L and justify the correctness of your regular expression.

   Solution:
   $R = (1+2)^*0((0(1+2)^*0) + (1+2))^*$

   Proof. We want to show that for any string s generated from R, $s \in L$. And for any string $s \in L$, s can be generated from R.

   - For any string s generated from R, $s \in L$:
     Observe that all strings generated with R can be written as $(1+2)^i0((0(1+2)^j0) + (1+2))^k$ where $i, j, k \geq 0$. Then the number of 0's generated by R is a function: *number of 0's* $= 1 + 2 * k'$ where $k' \leq k$, which is an odd number, so any strings s generated from R is also in L.
   - For any string $s \in L$, s can be generated from R:
     It is easy to see that the prefix of s up to the first 0 can be generated by $(1+2)^*0$. If s has more than one 0, consider the substring starting from the character after the first 0 to the third 0, $(1+2)^i0(1+2)^j0, i, j \geq 0$ can be generated from $((0(1+2)^*0) + (1+2))^*$. By repeatedly applying this strategy, we can use the RE to generate the prefix of s up to the last 0. Lastly we can apply $((0(1+2)^*0) + (1+2))^*$ again to generate the suffix of s starting from the last 0.

   □

10. **Properties of Regex:** Is the sentence S below true? If your answer is YES, then provide your reasoning as to why the sentence is true. If NO, provide a counterexample.

    S: Any language represented by a regular expression containing the Kleene star operation must necessarily contain the empty string.

    Solution: NO. Here is a counterexample:

$$0^*0$$

11. **Properties of Regex:** Is the sentence S below true? If your answer is YES, then provide your reasoning as to why the sentence is true. If NO, provide a counterexample.

    S: Any language represented by a regular expression containing the Kleene star operation must necessarily be an infinite language.

    Solution: NO. Here is a counterexample:

$$\lambda^*$$

12. **Regex to Language:** Write the language represented by the following regular expression $R = a^*b^*$ in set-theoretic notation.

    Solution: The set-theoretic representation of the language L of R is as follows:

$$L = \{a^n b^m \mid n, m \geq 0\}$$

or

$$L = \{w | w \text{ is a string that doesn't contain substring “ba”}\}$$

Question 2: programming language concepts (40 points)

For each of the sub-questions below, provide a concise, correct, and complete answer. Each sub-question is worth a maximum of x points.

1. **Pointer Semantics:** Consider the following code snippet in C++. What is the output of the variables x and y?

```
1 int x = 0;
2 int y = 0;
3 int *p = &x;
4 int *q = &y;
5 int **pp;
6 pp = &q;
7 **pp = 20;
8 cout << x << endl;
9 cout << y << endl;
```

   Solution: The variable pp is a pointer to a pointer of type int. On line 6 pp is assigned the pointer to the pointer q (which in turn is pointing to the variable y). On line 7 of the code there is double deref resulting the value of the variable y being modified. Hence, the values stored in the variables x and y are:
   x=0
   y=20

2. **Pointer Semantics and Type-checking:** State the conditions under which the following C++ Boolean expressions evaluate to true at run time. The types for the variables x and y are deliberately not supplied below. (Note that there exists a type for each variable such that the code is valid C++.)

```
*(&x) == x
&(*y) == y
```

   Solution: $*(\&x) == x$ is always true, regardless of the type of x. The reason is that when you take first compute the pointer to x (i.e., $\&x$) and then deref it, the result is indeed the value of the variable x.

   However $\&(*y) == y$ evaluates to true at run time, only when y is a valid pointer type. By valid pointer type we mean that y must have been declared as a pointer to some datatype in the code. Otherwise, you will get a compile error. Further, if the pointer y has the value NULL, the behavior is technically UNDEFINED. In all other cases (i.e., y is a valid pointer type and is not NULL), the expression $\&(*y) == y$ evaluates to true.

3. **Basic Object Oriented Design and Data Encapsulation:** Consider the following C++ code snippet:

```cpp
class Base {
   private:
     int a;
   public:
      Base(){...}
      int* get_a() {
          ...
         return &a;
      }
};
```

Do you think this code is using the principle of data encapsulation properly? If your answer is YES, argue how and why the code is consistent with basic data encapsulation principles. If your answer is NO, then argue why not.

Solution: NO. The getter function `get_a()` returns a pointer to the private data in the class Base. This completely violates the idea of data encapsulation because now any external code can access and manipulate this data in a object of class Base.

4. **Semantics of Inheritance and Subtype Polymorphism/Virtual Functions in C++:** Jean recently learnt C++, and the concepts of inheritance and polymorphism. She immediately went ahead and implemented the following code.

```cpp
 1 class Cat {
 2    public:
 3       Cat(){...}
 4       void get_cat_type() {
 5          cout << "Hi, I am a generic cat" <<endl;
 6       }
 7 };
 8 class Tiger: public Cat {
 9    public:
10       Tiger():Cat() {...}
11
12       void get_cat_type() {
13          cout << "Hi, I am a Tiger, from the cat family" <<endl;
14       }
15 };
16
17 int main()
18 {
19     Cat *cat;
20     Tiger tiger;
21     cat = &tiger;
22     cat->get_cat_type();
23 }
```

She was expecting that her code on line 22 would print the string 'Hi, I am a Tiger, from the cat family'. Instead it printed "Hi, I am generic cat'. What went wrong? What principle should Jean use to fix her code to get expected behavior? How exactly would you implement this principle?

Solution: She forgot to use the principle of virtual functions. If she had declared the `get_cat_type()` as a virtual function in the base class, then at runtime the derived class version of this function would have been called at line 22. This is due to subtype polymorphism (same function name, but different behavior depending on whether the object is of the base or derived class).

The way virtual functions are implemented by C++ compilers is via the concept of runtime binding. That is, at compile time, the compiler will generate code for a virtual table (vtable) to be part of the Cat and Tiger objects. This table contains pointers to various virtual functions defined in the class Cat. At runtime, depending on the type of the object, the appropriate function pointer in the vtable will be looked up and jump to that function will be executed.

5. **Polymorphism vs. Inheritance:** What are the differences between polymorphism and inheritance? (also see my answer in Piazza post @328)

Solution: Inheritance is the most fundamental concept in the context of object-oriented programming, and it is primarily about data encapsulation and code reuse (think of the use of base class non-virtual member functions in derived classes).

Polymorphism can be added on in various ways to further enhance the value one can derive from inheritance. Examples include function overloading (ad-hoc polymorphism), generics or templates (parametric polymorphism), and class sub-typing and virtual functions (subtype polymorphism). All forms of polymorphism are conceptually about using the same function with different data types for greater code clarity. However, in practice, functions with the same name but different signatures (i.e., different argument and return types) have to be somehow instantiated (either via user-provided code or compiler-generated).

Ad-hoc polymorphism refers to using the same function name (e.g., add) with different argument and return types (signature of the function). The user typically has to supply the code for each signature of the overloaded function, and the compiler will make sure to use the right version depending on the input types at the function call site.

Parameteric Polymorphism or templates: We talked at length in class about templates. The basic idea behind templates is that for many types of functions or classes (think container classes), the code should be independent of the type of data stored. For example, the push/pop() functions of stack datatype need not be made aware of the type of data being stored on stack.

Subtype Polymorphism refers to the idea that we can assign a subtype (a derived class) to a variable of a super type (a base class) and the resultant code is type-correct. Further, languages that support this kind of polymorphism also provide support for some kind of virtual functions, i.e., member functions that have the same name in the base and derived class, but different functionality such that depending on whether a subtype variable is assigned to a super type, the appropriate function would be invoked at run time.

There are many different kinds of inheritance (e.g., multiple inheritance), just as there are many kinds of polymorphism. However, some programming languages, such as Java, limit the kinds of inheritance you can use to only single inheritance.

6. **Semantics of Inheritance and Constructors in C++:** Mary hastily wrote the following code and got a very unexpected compile error. Could you help find the error in her code?

```cpp
 1 #include <iostream>
 2 using namespace std;
 3 
 4 class base {
 5    private:
 6       base() {
 7          cout << "Hey: I am the base constructor()\n";
 8       }
 9 };
10 
11 class derived: public base {
12    public:
13       derived() {
14          cout << "Listen, I may be derived, but I am better than you\n";
15       }
16 };
17 
18 int main()
19 {
20    derived d;
21 }
```

Solution: You can't have a base constructor be private and simultaneously derive from it. The reason is that private functions are not accessible outside of the class in which they are defined (in this case class base), and when a derived class variable is instantiated on line 20, one of the tasks that the derived class constructor performs is to call the base class constructor. However, the compiler detects the fact that the base class constructor is private and not accessible to the class derived, and hence throws an error.

7. **Too Much Encapsulation:** After having written C code for years, and having learnt about object oriented programming recently, John excitedly embraced all things encapsulation. In particular, he wrote the following code. Is there something wrong with John's class design?

```cpp
class Base {
   private:
      int m;
      int get_m() {
         return m;
      }
   public:
      Base(){
      }
};
```

Solution: The trouble with John's code, as written, is that all ways of reading the variable are private. This means that only member functions can access the variable m. Typically, getter functions are public that allows any code that uses the API of class Base can at least read the data stored in class Base. Encapsulation is about controlling access to data and functionality, and not about denying access.

8. **Templates in C++:** In class we studied about function and class templates. Chunxiao recently wrote the following two pieces of code, both using function templates. He was very excited when the first piece of code worked correctly, but the second one gave a compile error. What is going on?

```
template <typename T>
T min(T a, T b) { return a < b ? a : b;}

int main()
{
    int x;
    x = min<int>(10, 3.14);
    return x;
}
```

Listing 1: code using templates

```
template <typename T>
T min(T a, T b) { return a < b ? a : b;}

class ADT {//Abstract DataType
 private:
    int value;
 public:
    ADT(int v) : value(v) {}
    ...
};

int main()
{
    ADT x(10), y(4);
    y = min<ADT>(10, x);
    return y;
}
```

Listing 2: code using templates and an ADT

Solution: For any function template, at compile time we have to provide the actual function for every type t with which the function templates is instantiated at a call site in the code (i.e., `F<t>`). The only exception to this rule is if the function is already defined. For example, in code listing 1 above, where the comparison function $<$ is already defined for the instantiated data type int as part of the C++ programming language. By contrast, the comparison function $<$ is NOT defined for the type ADT and hence the template function call `min<ADT>` is undefined in code listing 2, resulting in a compile error.

9. Can a program P written in an object-oriented language (say, C++) be converted into an input-output equivalent program Q in a language that is NOT object-oriented. By input-output equivalent we mean that given any input I, the output of P and Q match. (We don't care about differences between P and Q in terms of memory or other system-level behavior.)

   Solution: Yes, of course. For example, your typical C++ compiler translates your C++ code to an input-output equivalent C code and then to assembly code, both of which are clearly not object oriented. Typically, object oriented abstractions are implemented using a heavy use of pointers at the C-level.

10. **pass by reference vs. pass by value vs. pass by pointer in C++:** Consider the following code snippets. Are the values of i, j modified after each of them is executed? If so, explain why. If not, explain why not.

```
template< typename T >
void my_swap(T& t1, T& t2) {
    T tmp(t1);
    t1 = t2;
    t2 = tmp;
}

int i = 4, j = 3;
int& iref = i, jref = j;

my_swap(i, j);
my_swap(iref, jref);
```

Listing 3: pass by reference

```
template< typename T >
void my_swap(T t1, T t2) {
    T tmp(t1);
    t1 = t2;
    t2 = tmp;
}

int i = 4, j = 3;
my_swap(i, j);
```

Listing 4: pass by value

```
template< typename T >
void my_swap(T& t1, T& t2) {
    T tmp(t1);
    t1 = t2;
    t2 = tmp;
}

int i = 42, j = 10;
int* pi = &i, *pj = &j;
my_swap( pi, pj );
```

Listing 5: pass by pointer

Solution: In listing 3, the values of the variables i and j in the caller function that calls `my_swap()` will be swapped at lines 11 and 12.

In listing 4, look carefully at the signature of `my_swap()`, where the function is defined with 'pass by value'. Here, the values will not be swapped at line 9.

In listing 5, the pointers will be swapped, but not the values. That is, pi will point to j after the call to `my_swap()`, while pj will point to i.

Question 3: Processes, System Calls/Signals, Control-hijack Attacks, Compiled vs. Interpreted Languages (30 points)

Each of the questions below are worth x points. Please provide a pithy, precise, correct and complete answer for each sub-question.

1. **Integer Overflow:** Is there an integer overflow in the following code? Can it result in a buffer overflow? If your answer is YES that there is an integer overflow in the code below and it can cause a buffer overflow, you have to argue why this is the case. If your answer is NO, you have to argue, why not. (Note: Not all integer overflow errors may cause a buffer overflow. And we assume int type is 32 bits.)

```
1    ...
2      int nresp = packet_get_int(); //read nresp number of bytes from input stream
3      if (nresp > 0) {
4          response = malloc(nresp*sizeof(char*));
5          for (i = 0; i < nresp; i++) //read input stream
6              response[i] = packet_get_string(...);
7      }
8      ...
9  }
```

Listing 6: Code Fragment from OpenSSH

The precise and detailed semantics of `packet_get_int()` and `packet_get_string(...)` are irrelevant to answering the question. The only pieces of information you need are the following, namely, that `packet_get_int()` returns an 32-bit integer, and that `packet_get_string(...)` returns a string controlled by the attacker.

Solution: Yes.

First observe that there is a possibility of integer overflow due to the multiplication call to malloc at line 4. If the variable nresp has the value 0x40000000, then nresp*sizeof(char*) is guaranteed to have an integer overflow. This means that malloc is being asked to allocate a 0 byte buffer on line 4. Therefore, when the attacker-controlled string returned by `packet_get_string()` is written into the buffer response, a buffer overflow can occur (depending on the length of the attacker string). This is potentially exploitable.

2. **Semantics of fork():** Consider the series of fork() calls in the following piece of code

```
 1   ...
 2 int main(void) {
 3   std::cout << "my pid: " << getpid() << std::endl;
 4   //      << "my parent pid: " << getppid() << std::endl;
 5 
 6   char alpha;
 7   int status;
 8   int id = fork();
 9   int id1 = fork();
10   if (0 == id || 0 == id1) {
11     std::cout<<"hi, I am a child process: " << getpid() << " and my parent process
       is: " << getppid() <<std::endl;
12     alpha='a';
13   }
14   else {
15     std::cout<<"hi, I am a parent process: " << getpid() <<std::endl;
16     alpha='f';
17     wait(&status);
18   }
19 ....
20 }
```

Listing 7: fork calls

What is the total number of processes created by the two fork() calls (including the parent process)? Draw a graph or textual representation of the parent-child relationship between all the processes thus created. In general, if you have n fork() calls, how many total number of processes will be created?

Solution: A total of 4 processes will be created including the parent process, when the code above is done executing line 9. In general, if there n fork calls, a total of 2^n processes will be created.

To see this better, let P denote the parent, and C denote the first child on line 8. Under normal circumstances, note that line 9 will be executed in both processes P and C. Thus, two more children C1 (the second child of P) and C2 (the child of P) will be created. The relationship between them is:

```
P --> C --> C2
P --> C1
```

3. **Process Memory Layout and Call Stack:** How does the call stack of a typical process work during its execution such that control-flow of the program follows sequentially in the order in which the code is specified by you? Recall that whenever your favorite programming language compiler is compiling a function call site a jump instruction is used to implement a jump to the code of the function. Hence, the call stack to ensure that your program's control reaches the line right after the function call in the sequential (top-down) layout of your code.

   To assist you with answering this question, we have provided some pseudo-code below. Use the code line numbers to describe how the program's control reaches the line of code immediately following the function call site, once a function completes its execution.

```
 1 f(...) {
 2   int * x = malloc(...);
 3   //process data pointed to by the ptr x
 4   ...
 5 }
 6
 7 int main(void) {
 8   f();
 9 ....
10 }
```

Listing 8: Call Stack

Solution: At compile time, the compiler injects code at all call sites to manage the call stack at run time.

At run time, upon invocation of the function f() at the call site on line 8, this compiler-injected code is executed to create an entry on the call stack corresponding to f(). This entry contains local variables, certain registers, pointers to exception handlers, as well as the return address (ret-addr), i.e., the line of code right after line 9 where the control must be transferred to, immediately after the execution of f().

During the execution of f(), a malloc() call is made at line 3. In this event, the execution of f() is suspended right prior to the malloc() call, a stack frame is created, populated, and pushed on to the call stack corresponding to malloc(), and a ret-addr corresponding to the code on line 5 is entered into this stack frame. This ensures that once malloc() completes its execution (and the corresponding stack frame has been popped), control can reach line 5. An important point to remember is that all register values corresponding to f() that were stored are now restored as well such that f() can resume execution.

4. **Heap-based Overflow Attack:** In class Anka studied the possibility of a heap-based overflow attack. In this style of attack, the victim program has a buffer adjacent to a function pointer or virtual table on the heap. Further, this buffer can be written into with an arbitrary string by an attacker via a port or input and this buffer is susceptible to an overflow (i.e., there are no length checks or other defenses). By writing into such a buffer, the attacker can modify the function pointer to point to code on the heap that is controlled by her. (Note that the attacker's attack string contains both the address on the heap where to jump to, as well as the payload.)

   Anka decides to lift the idea of NX (not execute) bit, implemented as part of the call stack memory pages, and apply it to the heap. That is, if the data stored on the heap are interpreted as executable code and executed upon, then it would cause the OS to send an SIGABRT (abort) signal to such a process. The reason this method prevents heap-based code injection attacks is because now the attacker's code on the heap is rendered non-executable.

   However, there is a problem with this mechanism that affects virtual machines such as JVM etc. Could you identify the issue? **Hint:** Think in terms of data in the memory (heap) of a virtual machine or interpreter such as JVM that can be interpreted as code.

   Solution: The problem with Anka's solution is that virtual machines such as JVM will not be able to interpret data as code, a necessary feature of all modern VMs.

   All modern VMs, such as the JVM, have a JIT (Just-in-Time) compiler, that takes pieces of user program (as data stored on the JVM's heap) and generates compiled machine code from it (again stored on the heap). It is this machine code that is finally executed by the system. In order to be able to do this, the code produced by JVM's JIT is stored on the heap and executed.

5. **ROP Attacks and ASLR:** In class we studied Address Space Layout Randomization (ASLR), a really interesting and clever defense mechanism against Return-Oriented Programming (ROP) attacks. Explain the basic principle behind ROP attacks and how ASLR can mitigate such attacks?

   Solution: Traditional code injection attacks generally require the following conditions to hold in order to be successful:

   (a) There exists some input or port via which an attacker can send a string
   (b) There exists a vulnerable buffer which can be written into by an attacker-controlled string S
   (c) There are no length checks (or ineffective length checks) on the input S, when it is being written into the vulnerable buffer B
   (d) A write into the vulnerable buffer B can result in a function pointer (e.g., pointers in a virtual table or exception handlers) or ret-addr be overwritten
   (e) The attacker's string S has a carefully crafted executable code in it
   (f) At the opportune time (e.g., when the victim function is done executing and the overwritten ret-addr is loaded into the program counter), the overwritten ret-addr or function pointer is used to jump to attacker controlled code on stack or heap

By contrast to the above, in an ROP attack steps 1-4 are similar, but instead of jumping to an attacker controlled string, control jumps to a gadget (snippets of victim program's code that can be strung together to form an attack). In this setting, the attacker does not need to inject code into the memory of the victim program and hence is much more difficult to detect and prevent.

At a high-level, the ASLR mechanism randomizes the layout of the memory of the victim program, including shared libraries, thus preventing the attacker from knowing the precise address of the first gadget to jump to in order to execute a successful attacks. There are details such as Position Independent Executable (PIE) code or high entropy in 64 bit machines (and how that is precisely achieved) that we didn't discuss that are crucial to the success of ASLR. In any event, ASLR has become a sophisticated method that is now standard in most Linux, Windows and other popular OSes.