FORMAT STRING ATTACK

A format string attack exploits the fact that functions such as printf() can accept a variable number of input arguments and the exact number of input arguments is implicitly specified in the first argument, the format string argument. These functions are often referred to as variadic functions. Given a call to a variadic function, the callee has no way of determining if the input format string specifies more input arguments than is set up by the caller; only the caller can check if the number of arguments it prepares is consistent with that specified in the format string. If an attacker can control the format string argument of a variadic function in a victim application, it is possible for the attacker to read or write the application's address space. If the caller of a variadic function can explicitly specify the number of input arguments it prepares as another input argument, it would have made format string attack much more difficult, if not impossible. Unfortunately the interface to existing variadic functions such as printf() does not permit this extension.

Format string vulnerability mainly stems from a programming error that leads the format string of a variadic function such as printf to be directly controllable by an external input. Consider the following two calls to printf: (1) printf ("%s", user string) and (2) printf (user string), where user string is derived from an external input. The results of these two calls are exactly the same if user string is just a simple character string. However, if user string contains conversion specifiers, each of which corresponds to a separate input argument and represents a command that controls how printf operates on its corresponding argument, the second call could trick printf into believing there are more than one input argument in this call. As a result, even though printf (user string) contains only one argument, the fact that user string contains K conversion specifiers is enough to convince printf to access additional memory locations on the stack that are beyond user string. If each of these K conversion specifiers in user string denotes an integer, then the attacker can trick printf into displaying the next K integers on the stack. To modify the victim application's address space, format string attacks exploit a special conversion specifier "%n", which counts the number of characters written so far and writes the result into the address given by its corresponding argument. By carefully crafting a format string that prints out a pre-computed number of characters before the "%n" specifier, the attacker can write a chosen value to some memory location. Moreover, because the additional arguments required by the conversion specifiers come from the stack frame of the caller to printf (user string), as does user string, they are likely to be controllable by the attacker as well. With the "%n" specifier and the ability to manipulate its corresponding argument, the attacker now has the

ability to write an arbitrary value to a chosen memory location. If the memory location chosen to be overwritten is a control-sensitive data structure that contains a return address or function address, the attacker could hijack the control of the victim program.

Denial of Service

In this case, when an invalid memory address is requested, normally the program is terminated.

Printf("username");

The attacker could insert a sequence of format strings, making the program show the memory address where a lot of other data are stored, then, the attacker increases the possibility that the program will read an illegal address, crashing the program and causing its non-availability.

printf (%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s);

CODE INJECTION ATTACKS ON HTML5-BASED MOBILE APPS

It is well known that the Web technology has a dangerous feature: it allows data and code to be mixed together, i.e., when a string containing both data and code is processed by the web technology, the code can be identified and sent to the JavaScript engine for execution. This feature is made by design, so JavaScript code can be embedded freely inside HTML pages. Unfortunately, the consequence of this feature is that if developers just want to process data but use the wrong APIs, the code in the mixture can be automatically and mistakenly triggered. If such a data-and-code mixture comes from an untrustworthy place, malicious code can be injected and executed inside the app. This is the JavaScript code injection attack. A special type of this attack is called Cross-Site Scripting (XSS), which, according to the OWASP top-ten list , is currently the third most common security risk in web applications. The Overview The decision to use the web technology to develop mobile apps opens a new can of worms, making it possible for the code injection attack to be launched against mobile apps; this is much more damaging than the XSS attack on web applications, simply because we give too much power to the apps installed on our mobile devices. Moreover, in the XSS attack, the channel for code injection is limited to web application server, which is the only channel for untrusted data to reach their victims. There will be many more exploitable channels in the code injection attacks on mobile apps. A common characteristic of these channels is that they all link the mobile devices to the outside world, essentially allowing the attacks from another device (not

necessarily a mobile device). Since smartphones constantly interact with the outside world, in addition to the traditional network channel, there are many new channels for untrusted data to enter mobile devices. For example, 2D barcodes, RFID tags, media files, the ID field of Bluetooth devices and Wi-Fi access points, etc. Malicious code can be embedded in data coming from these channels. If the code mixed in the data does not get a chance to be triggered, there is no risk caused by the code. That is why apps written using the native language are immune to this type of code injection attack. For example, even if attackers can embed a Java code inside a 2D barcode, there is not much chance for the code to be triggered mistakenly. This is not true for the HTML5-based apps, due to the dangerous features of the web technology. In particular, it is quite common for apps to display the data coming from outside, such as displaying the information in a 2D barcode. A number of APIs in the web technology are quite “smart”: they can separate the data from code, send the data to the HTML rendering engine and the code to the JavaScript engine, regardless of whether running the code is the developer’s intention or not. When the code gets executed, it can leverage the permissions assigned to the app, and launch the attacks on mobile devices, using the “windows” on the WebView that is opened by the PhoneGap framework and the HTML5 APIs.

ARRAY OVERFlOW

Important issues with “placement new” are:

1) ”placement new” allows any address allocated to the process to be used to place an object. The following code snippet illustrates the point: char c; int *b = new (&c) int;

2) ”placement new” does not enforce any bounds checking. Neither compile-time or runtime enforcement of bounds checking is applied.

3) Invocation of placement new does not carry out any type-checking. If memory is allocated to an instance of type T1, then placing an instance of type T2 at that memory succeeds even if T2 is not a compatible type of T1.

4) ”placement new” is used to populate an object or a data structure from a serialized instance (received from another source, possibly). However, since it does not enforce any checking of alignment, it may lead to incorrect semantics, and to program termination.

5) ”placement new” may lead to memory leaks.

A local variable might also be used to allocate memory, and can be used to carry out a stack overflow. Overflow of array-type buffers such as strings is commonly exploited by the attackers. Using ”placement new”, an attacker can mount such attacks in two steps. The idea is that a memory pool is already created and any new buffer needed is created out of that memory pool using ”placement new”. The constraint is that the size of the buffer is never greater than the size of the memory pool. In the first step of the attack, the attacker modifies the variable that stores the size of the buffer to a value larger than the memory pool size by overflowing an object using the appropriate method(s) specified above. The attacker would then be able to modify the data/bss/heap or stack locations that are beyond the memory pool. That way, code or arcs can be injected, return-to-libc-attacks can be easily carried out. In the next step, the user passes in a maliciously crafted string to the buffer as it is done in case of traditional buffer overflow scenarios - such as to open a remote shell. One can think such a vulnerability not to be so much dangerous as its exploitation requires two steps. However, we should keep in mind that for attackers, in order to take control of a system, a network or some sensitive data, making two steps is not a hard task. Now consider the following example. A function bool sortUname(char *uname) takes as input a list of user names separated by a newline characte, sorts them, and then adds them to the database, if they are valid. The number of user names n_unames is always less than the number of students n_students.

One example is

INFORMATION LEAKAGE

Information Leakage In this section, we show how information can be leaked via ”placement new”. A given memory pool is used to hold different objects or data structures at different program points. Information leak can occur when a smaller object is allocated in the memory pool, where a larger object was allocated earlier. The ”placement new” operator facilitates carrying out such operations, without however sanitizing the bits of the memory pool. If the attacker can control the size of second object/array, then information leak would occur. Consider the code in Listing 21, in which the password file is read into mem_pool; later a string from the user is stored in mem_pool. However, since the attacker (user) may pass a string that is of size less than SIZE (of mem_pool), the remaining part of the memory still has the contents of the password file

```
char mem_pool[SIZE]; char *userdata;

int main(int argc, char *argv[]) {

 //mmap/read a password file to mem_pool.

 //MAX_USERDATA <= SIZE

userdata = new (mem_pool) char[MAX_USERDATA];

//user input: userdata, sizeof(userdata) <= MAX_USERDATA [...]

//stores memory contents starting at userdata.

 store(userdata); }
```

OBJECT OVERFLOW

The fault based on “placement new” allows an attacker to place a larger object in the memory allocated to a smaller object. Placing a larger object in the same memory arena allocated for a smaller object leads to “object overflow”, that then overwrites certain memory locations. There are two ways to effect such a fault: (1) when the program constructs a larger object in the place of a smaller object, but does not carry out bounds checking on sizes, or (2) when the program accepts an object from another program (local or remote) or from network, and places it in the place of another program. In both cases, the smaller object is generally an instance of a supertype and the larger object is an instance of a subtype. A programmer may ignore checking the sizes before placement, as he might not think it is needed. Consider the following scenario: the “extra” members defined by the subclass would not be used and therefore would not be set by anyone, which why even if the size of new object maybe larger, it is harmless. If there is a way for the attacker or a malicious program to set some or all of the extra members to their chosen values, they can overflow the object in a “meaningful” manner. In case of (2), if the object is being created by an untrusted program such as a third party web service or a Javascript/AJAX/JSON2 , it can effect an attack on the receiving program. The programmer may not include any code to check the size because of the trust on the protocol based on which objects are received and sent from one program to another.

One example

Data/bss Overflow

In our example we show that an object of a subclass GradStudent (Listing 11) is used to populate the memory of the object of its superclass Student. However, note that an object of any class can be placed in the memory arena of an object of any class, or in fact starting at any address. GradStudent contains another member array, ssn[] that stores the SSN of a graduate student. In the code Listing 11, instances stud1 and stud2 are allocated in data/bss area (ELF format3) (precisely in the bss area as they are not initialized). Statement addStudent(false) creates stud2 as an instance of Student. Statement addStudent(true) creates stud1 as an instance of GradStudent. In the execution of the code segment if (isGradStudent) ..., the values of ssn[] are being set using attacker-specified inputs, thus changing the value of gpa in stud2.

```
Student stud1, stud2;
bool addStudent(bool isGradStudent) {
 GradStudent *st;
if (isGradStudent) {
//user input:ssn[0],ssn[1],ssn[2]; place st at &stud1
st->setSSN(...); }
else {
 //user input:gpa, year, semester; place st at &stud2
stud2 = new (&stud2) Student(gpa,...); }
}
addStudent(false);
 addStudent(true); //attack: overwrites ‘‘gpa’’ of stud2
```

USING STACK-OVERFLOW SOFTWARE VULNERABILITY TO CREATE A COVERT CHANNEL

main objective in this work is to develop an undetectable covert channel using the stack-overflow software vulnerability to hide traffic and secure data transfer.

use the stack-overflow vulnerability to construct and maintain a covert channel. First, we describe how Alice gets the delta mmap using an authenticated probing process. Then, we illustrate the operation of our covert channel with the specific example of sending a file. Getting delta mmap. Alice exploits the stack-overflow vulnerability in the server by trying to guess the correct address of a particular libc function (e.g., system). Alice continues to send attack strings (or probes) with guessed values until she successfully “hits” the libc function. Then, she computes the value of delta mmap We note that each unsuccessful guess by Alice causes the server’s child process handling Alice’s connection to crash and the parent process to fork a new child in its place with the same delta mmap . Each time Alice sends a probe, the server authenticates Alice; we call this process each-probe authentication. The each-probe authentication process proceeds as follows. First, Alice and server execute initialization process to produce a shared keySecond, Alice authenticates with the server, and the server replies with a message containing a nonce to defend against replay. The server then computes a message authentication code (MAC) of the nonce using HMAC keyed by the shared key produced in the initialization process. The server saves the generated MAC. Third, Alice receives the nonce, computes the HMAC using the shared key, sends a message to the server containing an attack string with a guess of delta mmap and the generated MAC. Fourth, the server receives Alice’s message and compares the stored MAC with Alice’s. If they match, Alice’s probe is passed to the vulnerable program; otherwise, Alice’s probe gets dropped. The second to fourth steps are repeated until Alice successfully guesses delta mmap. We note that Bob can distinguish the first attempt of Alice to guess delta mmap from subsequent guesses. Indeed, Bob can just consider the first successful guess as Alice’s initial attempt, or more sophisticated techniques (e.g., sending a start sequence over the channel) can be used. Sending a file. We illustrate the operation of OverCovert by describing its usage to transfer a file. Alice tries to guess the delta mmap of the vulnerable server program as described above. After Alice gets the correct value of delta mmap, she uses it to control the number of failed guesses. Alice reads the file to be sent in blocks of a certain size (e.g., 16 bits) and sends each block by making a number of failed guesses equal to the block’s numeric value (i.e., treating a 16-bit block as a 16-bit number) followed by a successful guess. More specifically, each time Alice wants to send a specific block, the initial value of the (guessed address of libc function) = (the base address of libc memory region) plus (offset of the function within the libc region) plus ((delta mmap) minus (value of the block to be sent)). For example, to send the block 0x0005, Alice needs to have exactly five failed guesses followed by a success; she starts her guessing attempts from five

less than the correct delta mmap, which Alice gets initially and which does not change as long as the sever is running. On other side, Bob monitors the server program and records the number of failed guesses performed by Alice before a successful guess.