Blind Format String Attacks

Fatih Kilic(✉), Thomas Kittel, and Claudia Eckert

Technische Universität München, München, Germany
{kilic,kittel,eckert}@sec.in.tum.de

Abstract. Although Format String Attacks (FSAs) are known for many years there is still a number of applications that have been found to be vulnerable to such attacks in the recent years. According to the CVE database, the number of FSA vulnerabilities is stable over the last 5 years, even as FSA vulnerabilities are assumingly easy to detect. Thus we can assume, that this type of bugs will still be present in future. Current compiler-based or system-based protection mechanisms are helping to restrict the exploitation this kind of vulnerabilities, but are insufficient to circumvent an attack in all cases.

Currently FSAs are mainly used to leak information such as pointer addresses to circumvent protection mechanisms like Address Space Layout Randomization (ASLR). So current attacks are also interested in the output of the format string. In this paper we present a novel method for attacking format string vulnerabilities in a blind manner. Our method does not require any memory leakage or output to the attacker. In addition, we show a way to exploit format string vulnerabilities on the heap, where we can not benefit from direct destination control, i.e. we can not place arbitrary addresses onto the stack, as is possible in stack-based format string.

Keywords: Security · Format string attacks

1 Introduction

Format string vulnerabilities are known for many years and are assumed to be easy to detect. But unfortunately there still exist applications, that are vulnerable to this kind of attack. According to the CVE database [17], the number of vulnerabilities that can be classified as format string vulnerability has decreased in the last 10 years. Over the course of the last 5 years, however, it appears to stay on a constant level.

These was, for example, a severe format string flaw in sudo versions 1.8.0 through 1.8.3p1 which was found in the `sudo_debug()` function (CVE-2012-0809). In Linux kernel through 3.9.4 existed a bug which allowed an attacker to gain privilege rights, which could be exploited by using format strings in device names (CVE-2013-2851). There also existed an exploitable format string bug in the Linux kernel before 3.8.4 in the function `ext3_msg()` which could be used to get higher privileges or crash the system (CVE-2013-1848). Therefore, we can

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2015
J. Tian et al. (Eds.): SecureComm 2014, Part II, LNICST 153, pp. 301–314, 2015.
DOI: 10.1007/978-3-319-23802-9_23

Table 1. Number of format string attacks in the last five years

Year	2009	2010	2011	2012	2013
Number	26	14	9	18	14

assume, that format string bugs will still be present in the future. Table 1 lists the number of registered format strings vulnerabilities over the last 5 years.

Since the first methods for a FSA were released, system wide protection mechanisms like Non Executable Bit (NX) and ASLR are implemented in many operating systems. Also compiler-based protections like stack-cookies and FORTIFY_SOURCE should protect from binary exploitation. All these protection mechanisms make exploitation more difficult nowadays. Nevertheless, Planet [13] has shown that FORTIFY_SOURCE can be circumvented and Payer et al. [12] have shown, that NX can also be bypassed.

All generic exploiting techniques shown in the past are relying on two mature constraints. First, the input buffer that is used by the attacker has to be placed on the stack, and secondly, the attacker requires knowledge about the output of the format string. In this paper, we instead assume, that the attacker is blind regarding to the output of the application. He will not receive any memory leakage by the exploited application. In addition, we also show that with our technique, the attackers payload may also be located on heap, instead of the stack.

In this paper we make the following contributions:

- We introduce a novel mechanism that enables an attacker to write to arbitrary memory locations using an FSA without the requirement to place the format string onto the stack.
- We describe a technique to redirect the control flow of an FSA vulnerable function in a blind fashion.
- We describe a Proof Of Concept (POC) implementation of our attack conducted with enabled protection mechanisms.

The rest of this work is structured as follows: First we introduce related work and various protection mechanisms in Sect. 2 and thereby motivate that format strings are still an issue in modern systems. Secondly, we describe the classical version of format string attacks in Sect. 3. We continue by showing our novel method to exploit format string attacks even without receiving any output in Sect. 4. In Sect. 5 we then present a POC that is able to use our method to execute arbitrary code on the victims machine. Finally we conclude our work in Sect. 6.

2 Related Work

The topic of FSAs is already known in the academic world for over a decade. The basic concept was first introduced by Newsham back in the year 2000 [11].

The concept was then extended and described in more detail in 2001 by Teso [14]. The attack has been enhanced by Haas [9] and Planet [13] in 2010. Haas is showing, that the memory leak of a format string can be used to calculate all relevant memory address to build the exploit string without any bruteforce, whereas Planet is showing a way to bypass the `FORTIFY_SOURCE` protection using format string attacks. We will describe the basic concepts of these attacks in the next section. In the recent years, however, this topic gained less interest. Payer et al. [12], recently describes a method for applying both Return Oriented Programming (ROP) [15] and Jump Oriented Programming (JOP) [5] to format string attacks described by Haas and Planet and also discusses different protection mechanisms.

We now describe protection mechanisms that have been established to mitigate memory write attacks. We hereby differ between two classes of protections mechanisms: Compiler-based and system-based protections. A commonly used compiler-based defense mechanism against control flow violations are stack cookies. The basic idea behind stack cookies is, that, in order to overwrite the return address of a function, a user has to overflow a buffer on the stack and thus overwrite everything between this buffer and the return address. If the compiler places a stack cookie between the buffer and the return address, the attacker also has to overwrite this cookie. As an attacker is unable to know the content of this cookie in advance, it is possible to detect the modification of the return address, if the cookie was overwritten by an attacker. This cookie can be circumvented by FSAs easily, because the place to be written can be directly controlled by the attacker. Another compiler-based protection mechanism is the compiler flag RELocation Read-Only (RELRO). This mechanism is resolving all addresses at the beginning and mapping the Global Offset Table (GOT) read-only, so an attacker cannot overwrite the function pointer and redirect the control flow.

A further mechanism that specifically protects against a FSA is using the `FORTIFY_SOURCE` option at compile time. The idea behind `FORTIFY_SOURCE` is to check the source code for the usage of certain insecure functions. These are common functions (e.g. `strcpy`) that use a given buffer and expect it to be delimited by a null terminator, which is not always the case. If the compiler detects the usage of such an insecure function (like `strcpy`), it tries to identify the size of the destination buffer and replaces the vulnerable function with a more secure function. A call to the `printf` function is replaced with a more secure function, so that the compiled program can handle a possible attack at runtime. If an attacker, for example, tries to use the `%n` parameter in a format string, the program will crash. Although this is a good idea, Planet has shown that this protection can be circumvented by overwriting the *IO_FLAGS2_FORTIFY* bit in the file stream by controlling the `nargs` value in the format string [13]. Another compiler-based protection is pointer encryption. This technique is used to encrypt instruction pointers with a simple encryption function which is not known by the attacker and thus prevents a pointer manipulation by the attacker [7]. This approach is thereby somehow similar to the stack-cookie approach. Although even if the algorithm uses XOR the attacker

can easily find the key if he knows a pair of plain and encrypted text. Furthermore, instruction set randomization uses the same idea in which the attacker does not know the instruction set [8].

After we now described some compiler-based approaches, we now look at common system-based mitigation approaches. One approach is ALSR. ALSR randomizes the memory addresses of both the executed code as well as the stack. Unfortunately, it only randomizes the prefix of entire pages, thus in case of $4K$ pages (which is common on the intel architecture), the last 12 bits of an address are not randomized. On modern systems we also see more randomization added to some mappings. They extend it to 20 bits and therefore only the last 4 bits are not randomized. This does not ensure security in 32 bit systems because the address can still be bruteforced. The reason for this is the limited number of randomized bits [1,3,4,10,16]. Another system-based protection mechanism is NX or Data Execution Prevention (DEP). Its goal is to hinder the execution of code that is located on a page that is supposed to contain data. Thus it hinders an attacker to prepare, for example his shellcode on the stack or heap.

Libsafe is a library which which can be used to protect against overwriting the stack at run-time. Equal to `FORTIFY_SOURCE` it replaces vulnerable functions like `*printf()` with secure versions. If there is an possible attack the library will kill the process and log the event. The disadvantage of this approach is that it works only for limited amount of functions [2].

FormatGuard is a patch for glibc which counts the arguments which are given to the printf function at run-time and compares it to the number of format specifiers (%). If the format string uses for more arguments then the actual number of printf arguments then a attack is assumed and the program will be terminated. To use this protection the programmer has to re-compile the program with FormatGuard. A problem with this approach is that it can only detect if the number of specifiers is changing but not if the variables are reordered. This kind of attack can not be recognized by FormatGuard [6].

3 Classical Attack

To give the reader a background in the topic of this work, we now describe the classical FSA attack that has evolved throughout the recent years. The classical FSA exploits the behavior of `*printf` functions, which are a class of functions that use formatting information to specify the format of the output. Since the `printf` function family are variadic c-functions, the number of format specifiers within the format strings is controlling the number of parameter which are used by the function and are thus popped from the stack[1]. The most important format specifiers for exploiting format string vulnerabilities are listed below:

%x - pop address from stack
%s - pop address and dereference
%n - write printed char count to address on stack

[1] e.g. *printf("id: %d, size:%d, name: %s",id,size,name)* consumes three arguments.

%hn - write to lower 16 bits (short)
%hhn - write to lower 8 bits (byte)

A basic format string vulnerability just passes a single argument to the *printf* function. This is illustrated in Line 5 of Fig. 1 were a user controlled buffer is given to the `printf` function as a single argument. In the classical exploit the buffer is defined as a character array on the stack. If the buffer contains user controlled input, an attacker can fill this buffer with arbitrary format specifiers, as listed above, and the function will access the next immediate value on the stack for each format identifier within the buffer. Depending on the used specifier, different actions will be executed on the stack. The attacker can, for example, shift the stack by using a *%x* operator or can dereference a memory address to access the content that is referenced by that address by using the *%s* operator. But the most important format specifier for having a generic way for exploiting this vulnerability is the *%n* operator. This specifier takes an address from the top of the stack, dereferences it and writes the total number of printed characters into the specified location. This allows an attacker to write arbitrary values to an arbitrary memory location, assuming that the vulnerable input buffer is located on the stack[2]. The chosen address could be the address of the saved return value, the address to an address in the GOT or an entry in the list of the destructors (*.dtors*). Thereby the attacker is able to change the control flow of the application, if she redirects such a pointer to some shellcode that she also prepared in advance.

4 New Attack Methods

After we discussed the classical FSA in the last section, we now describe a novel technique to apply an FSA even in an environment where the exploit string is placed on the heap and in addition, the user has no direct control over the stack content. Afterwards we will also describe, how it is possible to exploit this blindly, even without any feedback by the vulnerable program.

As this paper is about describing a blind FSA, we now want to define the term "Blind Attack": A blind attack is a network-based attack that is executed remotely without any local access to the attacked system. In addition, the attack does not require the attacking entity to receive any data from the attacked system. In the case of FSA this especially means that the output of the attacked *printf* function is not available to the attacker. Nevertheless, we assume, that the attacker is in possession of the executed binary beforehand. This is a legit restriction because most software is custom of the shelf software, that is not self developed and is available for the public.

[2] An input to a buffer like "$\backslash x78\backslash x4f\backslash x9e\backslash xbf$","$\%5u$","$\%10\$hhn$" will, for example, write the value $0x9$ to the least significant byte at the address $0xbf9e4f78$, because in this example the tenth value on the stack is containing our user input.

```
1  void logfunc(char *buf) {
2      char * pch;
3      pch = strtok (buf,"|");
4      while (pch != NULL) {
5          printf(pch);
6          pch = strtok (NULL, "|");
7      }
8  }
9  int parse(char *buf, int log) {
10     if (log == ENABLELOGGING)
11         logfunc(buf);
12     /* Do something using local stack variables */
13 }
14 int handle(clientsocket) {
15     char *buf = (char*)malloc(SIZE);
16     //...
17     recv(clientsocket, buf, SIZE-1, 0);
18     parse(buf,1);
19     free(buf);
20     //...
21 }
22 int func(serversocket) {
23     //...
24     while(1) {
25         pid = fork();
26         if(pid == 0) { /* ... */ handle(clientsock); /* ... */
27         }
28         //...
29     }
30     //...
31 }
```

Fig. 1. Format string vulnerability on the heap

4.1 Exploitstring on Heap

To exploit a FSA vulnerability, an attacker traditionally needs to store her attack payload in a buffer inside the vulnerable program. In Sect. 3 we have shown how a FSA is applied if the user input is saved on the stack. Within related work it is assumed, that it is required, that this buffer is located on the stack of the attacked system. This is an optimistic assumption, as is not always the case in practice. The problem with a heap based FSA is, that the attacker can only write to addresses which are already saved on the stack using the *%n* specifier. In this case the attacker can not place the destination address of the write on the stack to dereference it directly.

Stack-based FSAs, however, rely on user controlled input on the stack. The attacker places the exploit string, which contains the address of the write destination, directly inside the user input buffer. This address can then be directly accessed by the *$* operator or using the *%x* operator many times to pop all values

from top of stack until the attacker controlled data is at the top of stack. In our case we do not require attacker controlled input on the stack. This means only application controlled data is referenced on the stack. We therefore assume that there is no other input channel to place data on the stack, which would make the exploitation easier.

4.2 Arbitrary Write

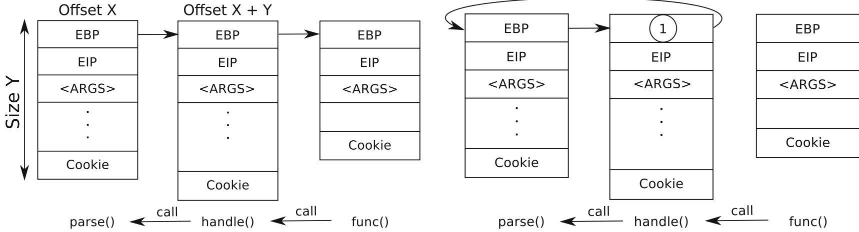
Above, we described how to write to application controlled locations by dereferencing the memory addresses on the stack and writing to it using the *%n* specifier. Now we focus on a generic exploitation concept to achieve arbitrary writes into application memory. The basic idea of our novel approach is to use the saved frame pointer, which is stored on the stack once a new function is called. If the application is not compiled with specific flags like *-fomit-frame-pointer* every function will save/push the last frame pointer on the stack in the prologue and restore it in the epilogue. We benefit from this fact because this address is always pointing to another location in the stack, which is also writeable. Therefore, no protection mechanisms like NX, stack cookies or ASLR will protect the system from an attacker writing to that location. Whenever an application is using the saved frame pointer feature, one frame pointer is pointing to the next frame pointer like a linked list. The next frame is therefore also located on the stack on higher addresses which can also be written to.

The goal of our mechanism is to use this list of saved frame pointers to achieve an arbitrary write to an arbitrary location within the system. With current FSA mechanisms we are only able to write to locations, which are already referenced on the stack of the current application. But by leveraging the linked list property of the saved frame pointers, we are able to modify the saved frame pointers on the stack according to our needs and thus achieve a situation in which we are able to write to an arbitrary location in memory. First we are using the saved base pointer (BP) on a lower address to overwrite the value of the next saved BP to point it to an arbitrary address like the GOT. In the second step we this location can be written to with an arbitrary value.

4.3 Changing the Control Flow

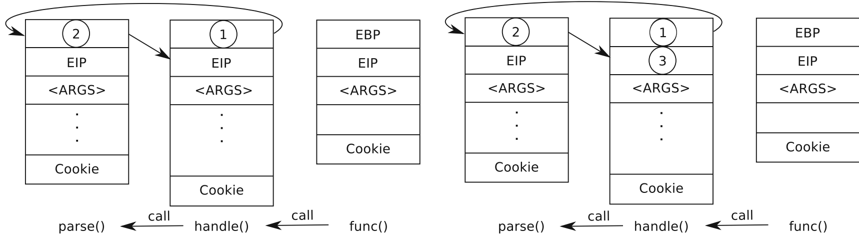
As we now are able to write to arbitrary memory locations, we describe how it is possible to hijack the applications control flow using an FSA. This still requires exact knowledge about the addresses, that have to be modified in order to control the execution flow. In the case of a blind attack, with no feedback from the attacked application and with ASLR activated at the same time, it is impossible to guess the exact address of our destination in advance. Entries like GOT are mainly at constant addresses but as mentioned in Sect. 2 the compiler flag RELRO will protect this locations from write access. In our approach we will only write to the stack, which is always writeable, to change the execution flow of the application. A generic way of controlling the execution flow is to overwrite the saved instruction pointer on the stack, so that an address is getting executed

on a *ret* command that was chosen by an attacker. As we already described above, the stack frames are connected with a linked list with directed pointers. Our goal is control the pointers in a way, that we can write to arbitrary locations on the stack.



(a) Initial Stack configuration with three functions.

(b) In the first step, the EBP of handle() is redirected to the EBP of parse().



(c) In the second step, the EBP of parse() is redirected to the EIP of handle().

(d) In the third step, the EIP of handle() is redirected to the attackers code.

Fig. 2. Sequence of overwrites to modify the return address

We will now describe our mechanism in more detail. To illustrate our mechanism first imagine a chain of three function calls like shown in Fig. 1. In this example a function `handle` calls a vulnerable function `parse` which in turn forwards the attackers buffer to an internal log function wrapper `logfunc`. This is a common scenario in both the Linux kernel and userspace applications. The initial stack layout of this scenario is depicted in Fig. 2(a). If we consider a format string like %6$*hhn*, we will write to the destination of the 6th value on the stack. The number six would be the *offset* in our explanation. The size is given as multiples of the architecture size, in our case 32 bit. *EBP* is the saved extended base pointer of the calling function. We do not have to care about the stack cookie protection, but if there is a cookie it will be at the bottom of the box, which we assume in our case as part of the frame content like the used stack variables. As the stack is growing to lower addresses, it is possible to overwrite the contents of the stack frames of the function `handle` and `parse` from within the log function. The attack consists of three format string overwrites, that use the pointers in *EBP*.

Note that the linked list of saved frame pointers is corrupted by this attack. An attacker may, nevertheless, reconstruct it after he is able to execute his own code, if it is required. This is only the case if the function is using local variables after the overwrite and before the return. Otherwise the application flow is changed and the attacker succeeded.

Table 2. Overview of required overwrites.

	Offset in format string	Dereferenced offset	Value written (address of)
1	X	X + Y	X
2	X + Y	X	X + Y + 1
3	X	X + Y + 1	Address of ROP gadget

In the first overwrite, the saved *EBP* of the function `handle` (1) will be modified to point to the saved *EBP* of the function `parse`. This is achieved by using the offset X in the format specifier and change the content at offset $X+Y$ with the address of offset X. Now we can directly address the saved *EBP* of `parse` as shown in Fig. 2(b). The next overwrite then replaces the contents of the saved *EBP* of `parse`, located at offset X (2) to the *EIP* of the function `handle`, located at offset $X + Y + 1$ by using the offset $X + Y$, as shown in Fig. 2(b). As a result of these first two steps an attacker generated a pointer on the stack, that points to the return address. In the third step the attacker overwrites this return address using offset X (3) to point to either the shellcode or some ROP chain, that the attacker prepared in advance. This final step is depicted in Fig. 2(c). After the vulnerable function finishes, the control flow will switch to a sequence of instructions that was chosen by an attacker. An overview over the conducted overwrites is given in Table 2.

4.4 Pointer Modification with ASLR Enabled

In our approach we leverage the saved frame pointer feature as it contains pointers that can be used during an FSA. In case the attacked system has stack ASLR enabled, an attacker is unable to guess the address he has to write to the stack. Unfortunately in its simple version, ASLR is not randomizing the whole address. For example all addresses inside a page will be constant, as ASLR only randomizes the beginning of the stack on page granularity. This means that effectively the least significant 12 bits, we assume a page size of $4K$ as described in Sect. 2, of the address will be constant and not randomized. In the case of an FSA an attacker can benefit from this behavior as he only needs to overwrite the least significant bytes of the frame pointer and redirect it to another frame pointer. Thus she modifies the least significant bytes of an address that is already pointing to the right location. Depending on the frame size the attacker has to overwrite one or two bytes. In the case of a good alignment and a distance less than 256 Bytes, an attacker does not need to care about ASLR, because only one byte

write is required. We can only write a multiple of eight bits using the $\%n$ operator. This means that in case of a two byte overwrite, four bits of randomization are overwritten by the FSA. In a practical exploit this is not a problem and an attacker is able to brute force these four bytes because of two reasons. First, if we have a network related application, each connection is transfered into its own process. This feature can be leveraged in a way that the attacker is able to crash the process without crashing the whole application. In case the exploit is not successful, the attacker can simply reconnect and try again. The second reason is, that only four bits of ASLR randomization is not a barrier for an attacker in this case, because the value only has to be found once. Every other write will also be at the same randomized four bytes and can thus be calculated beforehand. Note that as the connection handler is forked for every connection the stack will also be at the same address until the main application is restarted. In contrast to the 12 bit ASLR randomization, some systems use 20 bits of randomization for the stack. In this case an attacker has to bruteforce more bits, but as we will show this case in our POC, even with 20 bits of randomization the attack is practicable in a short time.

5 Proof of Concept

After we have introduced a novel technique to change the control flow of an application in a blind way using an FSA, we now introduce our POC implementation. In the following we assume the attack to be conducted on a 32 bit Linux system on the x86 architecture. In our tests we used an Ubuntu 14.10 system with the latest security patches applied. Therefore we assume that our binary is compiled with gcc in version 4.8. As already described, our vulnerable application consists of a networking daemon that forks a new process once it receives a new network connection. Each connection is than handled inside the newly created process. Our test system has the following protections activated: ASLR for stack, heap and libraries, NX on stack and heap, and RELRO. The stack addresses, where the BP are stored is randomized with 20 bit. After a local analysis of the attacked binary we will get the following values for the stack frame sizes: $X = 48$ Bytes and $Y = 48$ Bytes. This means that we will only require a one byte write if the least significant byte (LSByte) of the saved BP of *handle()* is between $0x60 (= X + Y)$ and $0xfc (= 0x100 - 4)$. Otherwise it has to be a two byte write. As it is the more complex case, we will only consider the case of two byte writes and show, that this technique is feasible even with bigger frame sizes.

First of all we will start with a simple bruteforce using three phases: In Phase 1, we will iterate over all possible values for the LSByte and restore the saved BP of *handle()*. Since the addresses on the stack are 32bit aligned, there are only 64 possible values for the LSByte. If the value that is currently checked does not match, we assume that the application crashes and the server socket is closed. We can recognize this behavior once we do not get any feedback. On the other hand we also have to take into account that not all successful tries imply a

correct guess of the correct LSByte. value. Thus after this step there can still be a number of false positives that we have to filter in the next step. Therefore, we will collect all checked LSByte. values that do not crash the server immediately in the first phase and verify them in the second phase.

In the second phase we reduce the number of possible values that we received in the first phase by verifying the integrity of those values. In our POC we designed four different verification tests that can be divided into two category. In the first category we try to rewrite pointers on the stack by building a chained list of pointers. For example, as we know the stack layout we can calculate the relative addresses of the saved frame pointers of other frames or any other variables within the frames and to overwrite their contents. In this case we do not expect the application to crash. In the second category we also use those addresses where we assume pointers on the stack and redirect the pointer chain to point to a non mapped memory location at the end, so *printf()* will crash during the memory write. After this verification process we have the exact address of the LSByte.

The third phase is then required to obtain the value of the second byte. Thus this phase is only needed if we have a two byte write. In this phase we are writing to the second byte, which has 256 possible values in total. Since we now modify the saved BP by a multiple of $4K$, the probability of having false positives is small. In our POC we did not get any false positives during our experiments. Our attack thus requires $256 + 64 + \delta$[3] connections in the worst case, which only takes few seconds in total. As the exploit strings used in this phase are small and can thus be executed very fast after *printf()* is called and the connections can also be multi-threaded, this step can be conducted in a short time.

After having the exact LSBytes of our address, we can now calculate all other stack addresses and build our exploit string to achieve an arbitrary write as describe in Sect. 4.3. The stack layout of our POC is illustrated in Fig. 3, where every column represents the stack layout in one of the described three stages of our attack. In Stage 3, we are overwriting the saved instruction pointer of *handle()* to return to a previously chosen destination. This destination could be the first ROP chain. Putting the whole ROP chain into the stack would assume, that we have enough space on the stack for all gadgets. It would also require more space in the input buffer or many calls to *printf()* for many writes using the format string vulnerability. Therefore the ROP chain should be located within the buffer itself and the number of the written gadgets by *printf()* should be small. It should only be used to switch the stack to the heap and to execute another ROP chain. But this technique has a big constraint. Since the *libc* is randomized, the non randomized gadgets are only available in the text section. We cannot guarantee, that we can find enough gadgets in the text section, especially if the binary is small. It has been proven that the libc gadgets are turing complete by Schacham [15], so we set our focus to use the libc gadgets here. As our technique is based on a remote connection, the Procedure Linkage Table (PLT) contains network related functions like *send()* and *recv()*. We are going to

[3] δ = false positive count * 4 (# of verification tests).

use this feature for constructing a memory leak and to extract the address of the libc back to the attacker. The addresses of the used library functions like *send()* are stored in the GOT at a constant and readable address. The call to a library function is done inside the text segment, which is not randomized. We can either call it by returning to the text segment or we can call it directly using the PLT entry, which is also on constant addresses. Overwriting the return value with *send@PLT* and leveraging the send function also requires that we now the value of *clientsocket*, to return the information to the right client. This value could be bruteforced, but in many cases it is stored on the stack. In our POC, for example, the clientsocket is a parameter of the *handle()* function. We are using a gadget to lift the stack to the position of clientsocket and return to *send@PLT* with the arguments *(clientsocket, send@GOT, 4, 0)*. This sends the address of *send@libc* to the attacker, who in the next step is able calculate all addresses inside the libc and build an exploit for a successful attack.

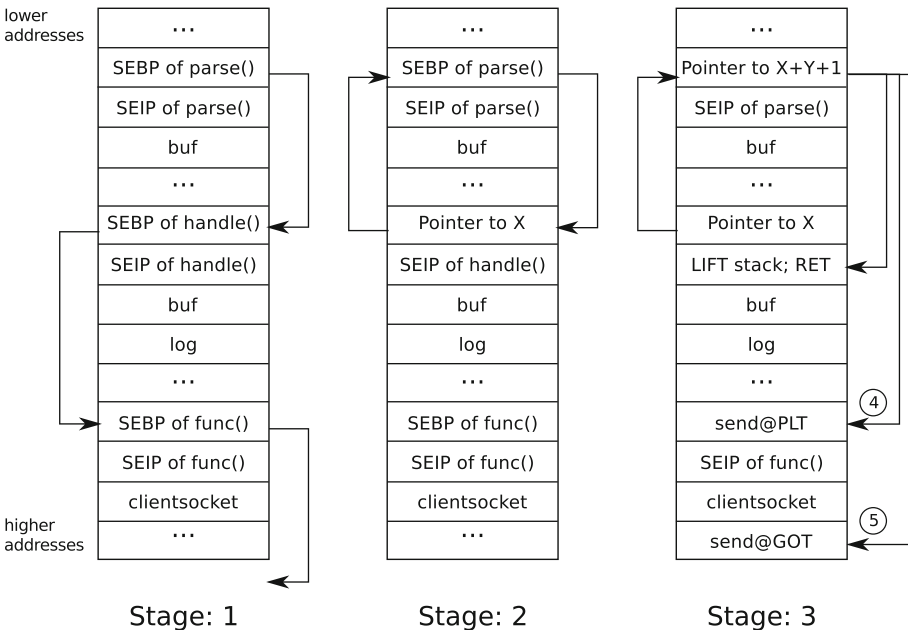


Fig. 3. Stacklayout for the proof of concept

6 Conclusion

In this paper we have shown, that format string attacks are still a security issue in recent history. We proposed a new approach, which does not require a memory leakage to exploit a format string vulnerability. Using our approach, we

can exploit an FSA blindly without having any output channel to the attacker or access to the local system. Our concept extends the classical FSAs to write to arbitrary memory locations even in cases where the format string is not stored on the stack but instead resides on the heap. We especially show that is is possible to redirect the control flow of an application using and modifying only pointers that are already present on the stack. We have also considered the most known protection mechanisms like ASLR, NX, RELRO and have shown that blind format string attacks are feasible even with activated protections.

References

1. Homepage of the pax team. http://pax.grsecurity.net/. Accessed 15 November 2013
2. Baratloo, A., Singh, N., Tsai, T.K.: Transparent run-time defense against stack-smashing attacks. In: USENIX Annual Technical Conference, General Track, pp. 251–262 (2000)
3. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium, vol. 120, Washington, D.C. (2003)
4. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: Proceedings of the 14th USENIX Security Symposium, pp. 271–286 (2005)
5. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, pp. 30–40. ACM, New York (2011)
6. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: Formatguard: automatic protection from printf format string vulnerabilities. In: USENIX Security Symposium, vol. 91, Washington, D.C. (2001)
7. Cowan, C., Beattie, S., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: Pointguardtm: protecting pointers from buffer overflow vulnerabilities. In: USENIX Security Symposium, vol. 91, Washington, D.C. (2001)
8. Gadaleta, F., Younan, Y., Jacobs, B., Joosen, W., De Neve, E., Beosier, N.: Instruction-level countermeasures against stack-based buffer overflow attacks. In: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems, pp. 7–12. ACM (2009)
9. Haas, P.: Advanced format string attacks. DEFCON 18 (2010)
10. Müller, T.: Aslr smack & laugh reference. In: Seminar on Advanced Exploitation Techniques (2008)
11. Newsham, T.: Format string attacks. Guardent Inc., September 2000
12. Payer, M., Gross, T.: String oriented programming. In: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW 2013. ACM (2013)
13. Planet, C.: A eulogy for format strings. Phrack magazine, 14(67), November 2010
14. Scut. Exploiting format string vulnerability. http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf
15. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 552–561. ACM, New York (2007)

16. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, pp. 298–307. ACM (2004)
17. The MITRE Corporation: Common vulnerabilities and exposures. https://cve.mitre.org/data/downloads/allitems.csv