Computer and Network Security
8th of July 2013

- This exam consists of 6 questions with subquestions. Unless indicated otherwise, every subquestion counts for 10 points.
- Mark every page with name and student number.
- Use of books, calculator, or additional course material is prohibited.
- Always explain your answers. At the same time, keep your answers short and to the point. Do not use pencil or red ink.

---

1. **True/False with Reason (10 points total).** For each of the statements below, state whether they are true or false. Explain your answer with a short justification, at most 2 sentences long.

   (i) Taint analysis propagates the taint bit of the source on *all* move/copy operations and *all* arithmetic operations. For instance, if `%eax` is tainted, the instruction "`movb %al,6(%ebx)`" would taint the byte at memory address `(6+%ebx)`. Likewise "`add %eax, %ebx` would taint `%ebx`, etc. (There must be ways to clean taint also, but they are irrelevant for this question.) We use the taint analysis to prevent malicious code from leaking sensitive information stored in a file: all bytes of the file are tainted when read and the system prevents tainted bytes from leaving the process in any way (written to disk, sent across the network, used in IPC, etc.).

   T/F: The taint propagation will effectively stop all leakage of the sensitive information.

   →Note: we do not care about the performance.

   (ii) **T/F**: In a C program, format string bugs alone cannot be exploited for control-hijacking attacks.

   (iii) An OS running on a new Intel processor marks the stack memory pages as non-executable using the NX bit feature[1].

   T/F : This mechanism prevents all attacks from getting a root shell by exploiting stack-based buffer overflows.

   (iv) **T/F**: The behavior of an application is only determined by the code being executed and the data being processed.

   (v) **T/F**: A ROP attack can be performed even when neither the version nor the load address of the C library is known.

   (vi) **T/F**: Consider an idealized, perfectly random hash function h, i.e., h is selected uniformly at random from all functions mapping $\{0,1\}^*$ to $\{0,1\}^k$. Given $y = h(x)$ for some x, the expected value for the number of attempts required to find any preimage x', such that $h(x') = y$, is on the order of $2^{k/2}$.

   (vii) **T/F**: The same-origin principle states that a websites's frame can access cookies set by the website and nothing else.

   (viii) **T/F**: A stateless firewall can perfectly block TCP connection initiation requests from an external location to any local host, while at the same time allowing returning traffic from connections initiated by local hosts.

2. **One Time Pad**

   In this problem, your task is to encrypt a message $M \in \{0,1,2,3,4\}$ using a shared random key $K \in \{0,1,2,3,4\}$. Suppose you do this by representing K and M using three bits (000, 001, 010, 011,

[1] Briefly, the NX bit (no-execute bit) is a flag added to every memory page indicating whether the contents of the page can be executed or not. When NX bit is set for a page, the processor never executes any byte in the page

100) each, and then XORing the two representations. Given a ciphertext 101, what can you infer about the corresponding plaintext? Do you think this scheme has the same security guarantees as the one-time pad? Explain.

3. **Memory errors**

   A server program is run SUID to allow it to do some necessary initialisation and shutdown. The server accepts messages from users in the following function:

```
1  void do_stuff (char *request) {
2       char buf [128];
3       strcpy (buf, request);
4       ... do something useful ...
5  }
```

   (a) Since the `request` comes from a remote party, the function `do_stuff()` is clearly vulnerable. As an attacker, can you: (i) manipulate the stack as is, (ii) manipulate the stack if there are stack canaries, (iii) execute injected shellcode, (iv) leak information about variables on the heap, (iv) manipulate the heap?
   → **Explain your answers with one sentence each. Assume no additional protection.**

   Unfortunately for you, the server drops privileges while handling requests. So it runs SUID in the in the initialisation and cleanup phase, but unpriviliged while handling the requests, as illustrated by the following pseudo code:

```
1   void server_loop () {
2        // First initialise (as root)
3        ... do what needs to be done as root ...
4        drop_privileges();
5
6        while (requests) {
7             req = receive_request();
8             do_stuff(req);
9        }
10       // now clean up (as root)
11       regain_root_privileges();
12       ... do clean up as root ...
13  }
```

   (b) Can you still spawn a root shell? If so, how?

   Also consider the following program. Assume that it is running in 32-bit x86 Linux with all protections (NX, ALSR, stack canaries, ...) disabled. Optimization is also disabled. Your shellcode is injected at address 0xbffffe01. All local variables are pushed onto the stack in the order they are declared. Variables on the stack are aligned on 32-bit boundaries, so all addresses are multiples of 4. No other alignment is enforced. All functions make a proper stack frame including a saved frame pointer and do not store any other temporary variables on the stack.

```
void process_cmd(char *buffer, char *cmd) {
   short index;
   index = atoi(cmd + 1);
   buffer[index] = cmd[0];
}

int main(int argc, char **argv) {
   short i, len;
   char buffer[40000];

   len = strlen(argv[1]);
   memcpy(buffer, argv[1], len);
   buffer[len] = 0;

   for (i = 2; i < argc; i++) {
     process_cmd(buffer, argv[i]);
   }

   printf("result: %s\n", buffer);
   return 0;
}
```

   (c) Which arguments would you provide to exploit the above program? Be precise.

4. **Shellcode** Consider the following C code that embeds assembly code:

```
1  int main(void)
2  {
3    __asm__(
4       "xorl %eax, %eax\n"
5       "pushl %eax\n"
6       "pushl $0x64692f2f\n"
7       "pushl $0x6e69622f\n"
8       "pushl $0x7273752f\n"
9       "movl %esp, %ebx\n"
10      "pushl %eax\n"
11      "pushl %ebx\n"
12      "movl %esp, %ecx\n"
13      "leal 0x4(%esp), %edx\n"
14      "movb $0xb, %al\n"
15      "int $0x80\n"
16   );
17 }
```

(a) What does the aforementioned assembly snippet do? Explain in terms of what the attacker is trying to achieve.

Note: an ASCII table and a syscall table can be found in the appendix

(b) Show the content of the registers eax, ebx, ecx, edx, and esp right before executing the assembly instruction "int $0x80". Also show the stack layout using symbolic values for addresses, when needed. For instance, assume the first push starts operating at the address ADDR.

5. **Network protection**

On a day, an IDS inspects 1.000.000 events. It raises 50 alerts, 10 of which turn out to be false alarms. The (real) total number of attacks on that day was 70.

(a) Draw the confusion matrix for the IDS.

(b) Calculate the precision, recall, F-measure and accuracy of the IDS.

(c) Explain the base rate fallacy (show the math using the above example!).

(d) Suppose we could deploy a mechanism that would ensure IP source addresses in IP packets correspond to the actual sender of a packet (i.e., it's impossible to "spoof" source addresses). For each of the following threats, explain whether (and briefly why) the mechanism would: (1) completely eliminate the threat, (2) reduce/eliminate part of the threat, but not all of it, or (3) have no impact on the threat.

(i) Buffer overflow attacks on a remote server.

(ii) TCP SYN fooding.

(iii) SMURF attacks.

(iv) Man-in-the-Middle attacks using ARP poisoning.

(v) DNS Cache poisoning

6. **Web attacks**

(a) Bored with CNS, you surf to your favourite news site. It lists headlines of major news events and when you click on the headline, it gives you the full news item. You click on the headline that reads "University professor sued for training hackers". The article is fascinating and inspiring. You also notice that the url reads: `www.clf.com/news?id=6`. For a laugh, you modify the query to: `www.clf.com/news?id=6cnssux--`. You are taken to a page that says: `Sorry, the article '6cnssux' does not exist!`

(b) Are there any indications that this site might be susceptibe to a XSS attack? If not, why not? If so, explain in detail how you would continue to try and exploit this site.

(c) Are there any indications that this site might be susceptibe to an SQL attack (i.e., does it look promising)? If not, why not? If so, explain in detail how you would continue to try and exploit this site.

(d) Are there any indications that this site might be susceptibe to a CSRF attack? If not, why not? If so, explain in detail how you would continue to try and exploit this site.

Appendix I: System Calls

```
syscall
number    prototype
-------   ------------------------------------------------------------------------
 01.    int sys_exit(int status)
 02.    int sys_fork()
 03.    ssize_t sys_read(unsigned int fd, char * buf, size_t count)
 04.    ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
 05.    int sys_open(const char * filename, int flags, int mode)
 06.    sys_close(unsigned int fd)
 07.    int sys_waitpid(pid_t pid,unsigned int * stat_addr, int options)
 08.    int sys_creat(const char * pathname, int mode)
 09.    int sys_link(const char * oldname, const char * newname)
 10.    int sys_unlink(const char * pathname)
 11.    int sys_execve(const char *filename, char *const argv[], char *const envp[])
 12.    int sys_chdir(const char * filename)
 13.    int sys_time(int * tloc)
 14.    int sys_mknod(const char * filename, int mode, dev_t dev)
 15.    int sys_chmod(const char * filename, mode_t mode)
```

The following registers are used to pass information to the kernel when performing a system call:

```
 %eax - system call number (from table above)   %esi - fourth parameter
 %ebx - first parameter                         %edi - fifth parameter
 %ecx - second parameter                        %ebp - sixth parameter
 %edx - third parameter
```

Appendix II: ASCII table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL