Software Security – The Dangers of Abstraction

Dieter Gollmann

Hamburg University of Technology, Hamburg, Germany
diego@tu-harburg.de

Abstract. Software insecurity can be explained as a potpourri of hacking methods, ranging from the familiar, e.g. buffer overruns, to the exotic, e.g. code insertion with Chinese characters. From such an angle software security would just be a collection of specific countermeasures. We will observe a common principle that can guide a structured presentation of software security and give guidance for future research directions: There exists a discrepancy between the abstract programming concepts used by software developers and their concrete implementation on the given execution platform. In support of this thesis, five case studies will be discussed, viz characters, integers, variables, atomic transactions, and double linked lists.

1 Introduction

Once upon a time, computer security was about access control, with authentication and authorisation as its fundamental components [12]. Internet security was about communications security. Strong encryption was the main tool to solve problems in this area. Today, attackers send malformed inputs to networked applications to exploit buffer overruns, or to perform SQL injection, cross-site scripting (XSS), or cross-site request forgery (XSRF) attacks. Access control and encryption are of little help to defend against these current threats.

Lesson: Security is a moving target.

Software security has become our main challenge. Software is secure if it can handle intentionally malformed input [11]. Networking software is a popular target as it is intended to receive external input and as it involves low level manipulations of buffers. Mistakes at that level can allow an attacker to circumvent logical access controls by manipulations at a "layer below" [9]. Web applications are a popular target. They are intended to receive external input and are written by a multitude of authors, many of whom have little security expertise.

1.1 Security and Reliability

Reliability deals with accidental failures that are assumed to occur according to some given probability distribution. The probabilities for failures are given first; then the protection mechanisms are constructed and arguments about their efficacy can be made. To make software more reliable, it is tested against typical usage patterns.

V. Matyáš et al. (Eds.): The Future of Identity, IFIP AICT 298, pp. 1–12, 2009.

> It does not matter how many bugs there are, it matters how often they are triggered.

In SQL injection attacks and the like, the attacker picks the inputs – and their probability distribution – with the aim to penetrate security controls. In security, the defender has to move first; the attacker picks his input to exploit weak defences. To make software more secure, it has thus to be tested against "untypical" usage patterns, but there are typical attack patterns.

> *Lesson: Measures dealing with failures that are governed by given probability distributions address reliability issues rather than security issues.*

Think twice about using reputation or "trust" for security! These approaches extrapolate future actions from past behaviour and do not capture strategic decisions by truly malicious attackers.

2 Dangers of Abstractions

When writing code, programmers use elementary concepts like character, variable, array, integer, list, data & program, address (resource locator), or atomic transaction. These concepts have abstract meanings. For example, integers are an infinite set with operations 'add' and 'multiply', and a 'less or equal' ordering relation. To execute a program, we need concrete implementations of these concepts.

Abstraction hides "unnecessary" detail and is a valuable method for understanding complex systems. We do not have to know the inner details of a computer to be able to use it. We can write software using high level languages and graphical methods. Anthropomorphic images explain what computers do (send mail, sign document). Software security problems typically arise when concrete implementation and abstract intuition diverge. We will explore a few examples:

- Characters
- Integers
- Variables (buffer overruns)
- Atomic transactions
- Double linked lists

2.1 Characters

To demonstrate the pitfalls when handling characters, we take a look at a failure of a standard defence against SQL injection attacks[1]. In SQL, single quotes terminate input strings. In a typical SQL injection attack, the malicious input

[1] See `http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string`; a similar problem in earlier versions of WordPress is discussed in `http://www.abelcheung.org/advisory/20071210-wordpress-charset.txt`

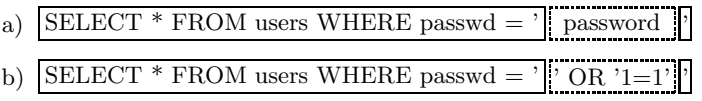

Fig. 1. Constructing SQL queries from strings, dashed boxes represent user input; case a) shows intended use; case b) is a SQL injection attacks that forces the WHERE clause to evaluate to true

contains a single quote followed by code segments picked by the attacker. When SQL statements are constructed by piecing together strings, some taken from the user input, others from the application issuing a database query, a single quote in user input can change the logical structure of the database query (Fig. 1 a)). Thus, attackers may be able to issue data base queries not envisaged by the application writer (Fig. 1 b)). As a countermeasure, the application could check user inputs and add a slash before any single quote encountered.

GBK (Guo Biao Kuozhan) is a character set for Simplified Chinese. In GBK, `0xbf27` is not a valid multi-byte character. When processed as single-byte characters, we have `0xbf` followed by `0x27`, a single quote. Adding a slash in front of the single quote gives `0xbf5c27`, but this happens to be the valid multi-byte character `0xbf5c` followed by a single quote. The single quote has survived!

> *Lesson: An operation may have different effects when observed at different levels of abstraction.*

2.2 Integers

In mathematics integers form an infinite set with addition, multiplication, and a "less or equal" relation. On a computer system, integers are represented in binary. The representation of an integer is a binary string of fixed length (precision), so there is only a finite number of "integers". Programming languages have signed and unsigned integers, short and long (and long long) integers. The operations on these data types follo9w the rules of modular arithmetic. With unsigned 8-bit integers we have $255 + 1 = 0$, $16 \cdot 17 = 16$, and $01 = 255$. With signed 8-bit integers we have $127 + 1 = -128$ and $-128/-1 = -1$.

In the following loop, the counter i has the value 2^k after the k-th iteration. At the level of the mathematical abstraction, the value of i will always be strictly greater than 0 and the loop would be infinite.

```
int i = 1;
while (i > 0)
{
i = i * 2;
}
```

Unsigned n-bit integers represent integers modulo 2^n. Hence, the value of i after n iterations is $2^n \bmod 2^n = 0$; there will be a carry-overflow and the loop will terminate. For signed integers, the carry-bit will be set after $n-1$ iterations and i takes the value -2^{n-1}.

In mathematics, the inequality $a + b \geq a$ holds for all $b \geq 0$. Such obvious "facts" are no longer true at the implementation level. Integer overflows can in turn lead to buffer overruns. Consider the following code snippet (from an operating system kernel system-call handler):

```
char buf[128];
combine(char *s1, size_t len1, char *s2, size_t len2)
{
if (len1 + len2 + 1 <= sizeof(buf)) {
    strncpy(buf, s1, len1);
    strncat(buf, s2, len2);
    }
}
```

Two character strings are concatenated and stored in a 128-bit buffer. In C, strings are zero-terminated so the program includes a check that should make sure that the buffer is large enough to hold both strings and the terminating zero. However, for 32-bit integers `len2 = 0xFFFFFFFF` results in `len2` + 1 = 0. If `len1` does not exceed the length of the buffer, the buffer will be written to while the number of bytes written can exceed the length of the buffer. The fact that computer integers do not behave like proper integers has led to vulnerable code more than once.

> *Lesson: Many programmers appear to view integers as having arbitrary precision, rather than being fixed-sized quantities operated on with modulo arithmetic [1].*

More information on integer overflows and on C libraries that properly handle finite precision integer arithmetic can be found e.g. in [11].

2.3 Variables

Variables are used in the abstract specification of algorithms. In the abstract specification we might denote the data type of a variable but we are not concerned with its actual representation. A buffer is the concrete implementation of a variable. If the value assigned to a variable exceeds the size of the allocated buffer, memory locations not allocated to this variable are overwritten. If the memory location overwritten had been allocated to some other variable, the value of that other variable can be changed. An attacker could change the value of a protected variable A by assigning a deliberately malformed value to some other variable B.

Unintentional buffer overruns crash software, and have been a focus for reliability testing. Intentional buffer overruns are a concern if an attacker can modify security relevant data. Attractive targets are return addresses specifying next method to be executed and security settings.

Historic Perspective. Since the contribution by Aleph One [15], buffer overruns have been extensively studied in the literature on software security, see e.g. [11,17,8]. We leave a detailed treatment of buffer overrun attacks to these sources and only give a brief historic perspective. Our first example from the 1980s relates to Digital's VMS operating system. The login procedure had the option of logging in to a particular machine by entering

username/DEVICE =<machine>.

In one version of VWS the length of the argument *machine* was not checked. A device name of more than 132 bytes overwrote the privilege mask of the process started by login. Users could thus set their own privileges. Our second example is the Morris worm from 1988 that exploited a buffer overrun in the fingerd daemon [7].

Lesson: Buffer overruns predate Windows.

For a recent case of a buffer overrun attack, we refer to a heap-based buffer overrun in μTorrent 1.6 allowing remote attackers to execute arbitrary code via a torrent file with a crafted announce header (CVE-2007-0927). μTorrent is a widely used lightweight torrent client. There is no automatic patching system and many of its users are "security-unaware" and do not use – or even disable – anti virus software. Hence, this case could have a higher damage potential than some operating system vulnerabilities.

Lesson: Buffer overrun attacks are moving to the application layer.

Defences against buffer overrun attacks come in various shapes. When developing code in la language like C, be careful and check how much you are writing to a buffer. The integrity of the return address can be protected by canaries [6] or by split control and data stacks [13,18]. The latter defences maintain the logical separation between code and data in the machine architecture. Shellcode insertion on the stack can be prevented by making the stack non executable. Finally, you can leave memory management to others and use a type safe language like Java.

Storage Residues. Buffer overrun attacks overwrite sensitive variables. There is a dual security problem, viz a process reading variables that it not yet had assigned a value to. In a multi-process system, several processes are running at the same time but only one is active. When a new process becomes active it gets access to resources (memory positions) used by the previous process. This is known as *object reuse*. *Storage residues* are data left behind in the memory area allocated to the new process. This is a security problem if sensitive data have been left. Operating systems thus usually allow a process only to read from memory it has written to.

To illustrate we summarize the Sun tarball story [10]. A *tarball* is an archive file produced by the `tar` utility. Some time in 1993 it was discovered that tarballs

produced under Solaris 2.0 contained parts of the password file. The following explanation emerged. The `tar` utility copied material in 512-byte blocks from disk to archive in a read/write cycle using a buffer. This buffer was not zeroed before data was read in. Thus, there could be a storage residue; if the last chunk of the file did not fill the buffer the previous content was read out. These memory positions happened to always hold a part of the password file.

This behaviour was caused by the following sequence of actions. During the read/write cycle `tar` looked up information about the user running the program. Therefore `/etc/passwd` was put on the heap. After checking the user the buffer for `/etc/passwd` was freed, but not zeroed. *tar* happened to be the next program getting this memory space, so memory residues were still there. The problem had not occurred in previous versions because the check of the user had happened earlier in the program. While fixing a bug, some code was removed and the vulnerability was exposed.

Are storage residues always a problem? Not so long ago, during a code review of Linux sources a read of an uninitialized variable was discovered in OpenSSL code. The offending line was commented out. After some delay in time, it was observed that the OpenSSL key generation algorithm produced predictable keys; the uninitialized variable had intentionally been used to provide randomness.

Lesson: In security, there are no correct answers.

2.4 Atomic Transactions

A *race condition* occurs when multiple transactions access shared data in a way that the overall results depend on the sequence of accesses. This can happen when multiple processes access the same variable. In multi-threaded processes, as in Java servlets, race conditions can occur between threads in a process.

A transaction is *atomic* if it is either executed in its entirety or if it has no effect at all. Access to a protected resource is fitting example for a transaction that should be executed atomically. The operating system first checks whether the access request is permitted; only in case of a positive outcome will the resource be made available to the requestor. If an attacker could change an essential parameter, e.g. a pointer to the resource, between those two steps, she could get access to a resource other than the one the initial check was performed for. Time-of-check-to-time-of use (TOCTTOU) is a well known security issue, as are `access()`/`open()` races in Unix [2].

For our illustrating example, we go further back in time to CTSS, one of the early time-sharing operating systems. One morning, users logging on to this system had the password file shown as the message of the day. The explanation was a race condition [5]. On CTSS, every user had a unique home directory. When a user invoked the editor, a scratch file with fixed name SCRATCH was created in this directory. At some point in time, the system was modified so that several users could work concurrently system manager. Later, the following occurred.

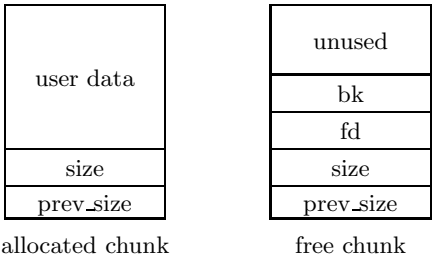

Fig. 2. Chunks in Doug Lea malloc

1. System manager A starts editing the message of the day, so SCRATCH in the system manager's directory contains this message.
2. System manager B starts editing the password file; now SCRATCH in the system manager's directory holds the password file.
3. System manager A saves the message of the day from SCRATCH, displaying the password file.

To defend against attacks exploiting race conditions enforce atomicity, e.g. through locks, so other processes are prevented from changing security relevant parameters. For more information on race conditions, on methods for scanning code for such vulnerabilities, and on possible countermeasures, see e.g. [4,14,3,16]. Finally, note that in Java it is the programmer's task to deal with race conditions by suitable synchronization of concurrent accesses.

2.5 Double-Linked Lists

There exist attacks more sophisticated than simple buffer overruns that exploit features of Unix memory management to overwrite arbitrary pointers. Our explanations will be based on Doug Lea malloc. Memory is divided into chunks. A chunk contains user data and control data. The control data include a boundary tag that gives the size of the chunk and the size of the previous chunk in memory. Chunks are allocated with `malloc()` and deallocated with `free()`. Free chunks are placed in bins. A bin is a double linked list, where chunks are ordered in increasing size. Free chunks contain boundary tags and forward and backward pointer to their neighbours in the bin (Fig. 2).

The size of a chunk is given in bytes, but chunk sizes are always multiples of 8 bytes. Thus, the three least significant bits of size are not used and have been designated for control flags:

- `0x1`: PREV_INUSE – indicates that the previous chunk in memory is free;
- `0x2`: IS_MAPPED
- Some libraries also use the third bit.

There should be no adjacent free chunks in memory. Hence, when a chunk is freed and a neighbouring chunk is free, both chunks are coalesced into a single chunk. Chunks are taken out of a bin with the `unlink` utility:

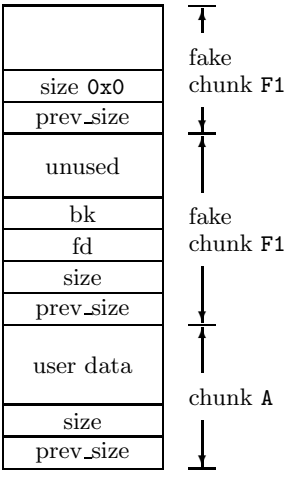

Fig. 3. Exploiting unlink after a buffer overrun

```
#define unlink(P, BK, FD)
{
[1] FD = P->fd;
[2] BK = P->bk;
[3] FD->bk = BK;
[4] BK->fd = FD;
}
```

`unlink` saves the pointers in chunk `P` to `FD` and `BK`. It then updates the backward pointer of the next chunk in the list: the address located at `FD` plus 12 bytes (offset of the `bk` field in the boundary tag) is overwritten with value stored in `BK`. Finally, the forward pointer of the previous chunk in the list is updated.

To demonstrate how `unlink` can be used to overwrite arbitrary pointers, we sketch a hypothetical buffer overrun attack [8]. Assume chunk `A` has a buffer overrun vulnerability; `A` is allocated. The attack is launched by overwriting the adjacent chunk `B` with fake chunks. These fake chunks are constructed so that there seems to be a free chunk next to `A` (Fig. 3).

Now free chunk `A`. The PREV_INUSE flag in chunk `F2` had been set so that `F1` is marked as free. `A` will be coalesced with the adjacent 'free' chunk and the fake chunk `F1` will be unlinked. Running `unlink(F1,FD,BK)` will add a 12 byte offset to the address given as the `fd` pointer in `F1`, overwriting this address with the value given as the `bk` pointer in `F1`. The attacker controls the values in `F1` and thus can overwrite a pointer of her choice with a value of her choice.

It is not necessary to have a buffer overrun to exploit `unlink`. To see how, we have to take a closer look at `free()`. Memory is deallocated with `void free (void *ptr)` where `*ptr` must have been returned by a previous call to `malloc()`, `calloc()` or `realloc()`. If `ptr` is null, no operation is performed.

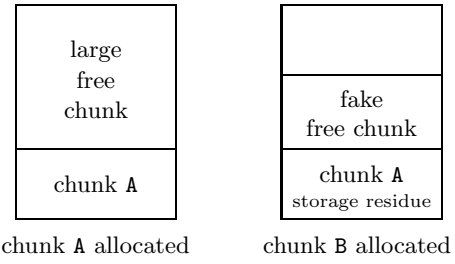

Fig. 4. Double free vulnerability

The behaviour is undefined if `free(ptr)` has already been called. Exactly this situation is the root of so-called *double-free vulnerabilities*.

Double free attacks exploit programs where memory is deallocated without setting the respective pointer to null. They only work if current memory usage is favourable to the attacker, but of course attackers can make their own luck. The vulnerable program allocates a memory block `A` that has to be adjacent to free memory (Fig. 4 left). When `A` is freed, forward or backward consolidation will create a larger block. Then the attacker allocates a larger block `B` hoping to get space just freed. In this case, a fake free chunk is written into `B` adjacent to the storage residue of `A` (Fig. 4 right). When `free(A)` is called again, consolidation with the fake chunk will overwrite a target address in the way described above. Double free vulnerabilities have been found in zlib (CA-2002-07), MySQL, Internet Explorer, Linux CVS, and MIT Kerberos 5.

Uninitialized memory corruption is a similar attack method. An exploitable vulnerability has been reported for the Heimdal FTPD server (CVE-2007-5939). In the code given in figure 5 `ticketfile` is declared but not initialized[2]. If `pw` is equal to null the program will jump to label `fail` and the uninitialized `ticketfile` will be freed. In this case the behaviour of `free()` is undefined and the attacker can try to manipulate the memory layout so that `free()` is applied to a pointer suitably prepared by the attacker.

We could treat double free and uninitialized memory corruption vulnerabilities as control flow problems. In the first case, memory deallocation is not performed completely; in the second case, memory allocation has not been completed before the memory is freed. The problems can be removed by tidying up memory allocation and deallocation.

We could also try to make `unlink` more secure. This utility is intended for taking elements out of a double linked list. The attacks violate this abstraction applying `unlink` to chunks that are not part of a double link list. As a defence – implemented e.g. in glibc 2.3.5 – one could check that the argument of `unlink` is part of a double linked list and meets other assumptions of Doug Lea malloc.

[2] See `http://archives.neohapsis.com/archives/fulldisclosure/2007-12/0175.html`

```
int gss_userok(void *app_data, char *username)
{
...
  if (data->delegated_cred_handle != GSS_C_NO_CREDENTIAL) {
    krb5_ccache ccache = NULL;
    char* ticketfile;
    struct passwd *pw;

    pw = getpwnam(username);

    if (pw == NULL) {
      ret = 1;
      goto fail;
    }

    ...

    fail:
    if (ccache)
    krb5_cc_close(gssapi_krb5_context, ccache);
    free(ticketfile);
  }
...
}
```

Fig. 5. Code segment from Heimdal FTPD

- check for membership in a double linked list locally with `!(p->fd->bk == p->bk->fd == p)`.
- Check if the first element in the bin is the one being added.
- Check if chunks are larger or equal to minimal size (16 bytes) and smaller than the memory allocated up to now.

3 Conclusion

Software security is not just about buffer overruns, and we have only just scratched the surface. There is more to it than just discrepancies between source code and object code, take integer overflows as an example. There are no quick fixes like avoiding unsafe C functions or by writing code only in type safe languages. Indeed, software security cannot be solved entirely at the level of the programming language. Programmers can make logical errors when the implementations of the abstractions they are using behave in unexpected ways. When security research tries to get ahead of the next reported vulnerability, it might well systematically compare programming concepts with their implementations.

When a problem area becomes known, tools and libraries can help dealing with the issues arising, but these tools and libraries have to be used. It is a technical challenge to develop useful and efficient tools. It is an organisational

and motivational challenge to get those tools adopted. This challenge is not made easier by the fact that the focus of attacks is moving from operating systems to applications. Cross-site scripting was the at number one in the 2007 OWASP Top Ten Vulnerabilities[3]. In the CVE database, cross-site scripting was at number one in 2005, and SQL injection at number two in 2006. There are better chances reaching the software experts writing systems code than reaching the many application experts writing application code.

Final lesson: Security research will stay in business ...

References

1. Ashcraft, K., Engler, D.: Using programmer-written compiler extensions to catch security holes. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, pp. 143–159 (2002)
2. Bishop, M., Dilger, M.M.: Checking for race conditions in file accesses. Computing Systems 9(2), 131–152 (1996)
3. Borisov, N., Johnson, R., Sastry, N., Wagner, D.: Fixing races for fun and profit: How to abuse atime. In: 14th USENIX Security Symposium, pp. 164–173 (2005)
4. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties. In: 9th ACM Conference on Computer and Communications Security, pp. 235–244. Springer, Heidelberg (2002)
5. Corbato, F.J.: On building systems that will fail. Communications of the ACM 34(9), 72–81 (1991)
6. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium, pp. 63–78 (1998)
7. Eichin, M.W., Rochlis, J.A.: With microscope and tweezers: An analysis of the Internet virus of November 1988. In: Proceedings of the 1989 IEEE Symposium on Security and Privacy, pp. 326–343 (1989)
8. Foster, J.C.: Buffer Overflow Attacks. Syngress Publishing, Rockland (2005)
9. Gollmann, D.: Computer Security, 2nd edn. John Wiley & Sons, Chichester (2006)
10. Graff, M.G., van Wyk, K.R.: Secure Coding. O'Reilly & Associates, Sebastopol (2003)
11. Howard, M., LeBlanc, D.: Writing Secure Code, 2nd edn. Microsoft Press, Redmond (2002)
12. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems 10(4), 265–310 (1992)
13. Lee, R.B., Karig, D.K., McGregor, J.P., Shi, Z.: Enlisting hardware architecture to thwart malicious code injection. In: Hutter, D., Müller, G., Stephan, W., Ullmann, M. (eds.) Security in Pervasive Computing. LNCS, vol. 2802, pp. 237–252. Springer, Heidelberg (2004)
14. Lhee, K.-s., Chapin, S.J.: Detection of file-based race conditions. International Journal of Information Security 4(1-2), 105–119 (2005)

[3] See `http://www.owasp.org/index.php/Top_10_2007`

15. Aleph One: Smashing the stack for fun and profit. Phrack Magazine, 49 (1996)
16. Uppuluri, P., Joshi, U., Ray, A.: Preventing race condition attacks on filesystem. In: SAC 2005 (2005) (invited talk)
17. Viega, J., McGraw, G.: Building Secure Software. Addison-Wesley, Boston (2001)
18. Xu, J., Kalbarczyk, Z., Patel, S., Iyer, R.K.: Architecture support for defending against buffer overflow attacks. In: Proceedings of the EASY-2 Workshop (2002)