- Systemnahe Software ist in vielen Fällen in Besitz von Privilegien und gleichzeitig im Kontakt mit potentiell gefährlichen Nutzern, denen diese Privilegien nicht zustehen.
- Daher muß bei der Entwicklung systemnaher Software nicht nur auf die korrekte Implementierung der gewünschten Funktionalitäten geachtet werden, sondern auch auf die umfassende Verhinderung nicht gewünschter Zugriffe.
- Dazu ist die Kenntnis der typischen Angriffstechniken notwendig und die konsequente Verwendung von Programmiertechniken, die diese zuverlässig abwehren.

- Das Werkzeug *pubfile* soll dazu dienen, Dateien im Verzeichnis *pub* unterhalb meines nicht-öffentlichen Heimatkataloges zur Verfügung zu stellen.
- So könnte *pubfile* übersetzt und in */tmp* öffentlich zur Verfügung gestellt werden:

```
theon$ id
uid=120(borchert) gid=200(sai)
theon$ gcc -Wall -o pubfile pubfile.c
theon$ cp pubfile /tmp
theon$ cat ~/pub/READ_ME
This is the file named READ_ME in the directory /home/borchert/pub.
theon$ /tmp/pubfile READ_ME
This is the file named READ_ME in the directory /home/borchert/pub.
theon$
```

```
theon$ id
uid=819(gast) gid=207(guest)
theon$ /tmp/pubfile READ_ME
/home/borchert/pub/READ_ME: Permission denied
theon$ cat ~borchert/pub/READ_ME
cat: cannot open /home/borchert/pub/READ_ME: Permission denied
theon$
```

- Im Normalfall operiert ein Programm mit den Privilegien des aufrufenden Benutzers.
- Das gilt auch dann, wenn das Programm einem anderen Benutzer gehört.
- Hier operiert */tmp/pubfile* mit den Privilegien von *gast*.

```
theon$ ls -l /tmp/pubfile
-rwxrwxr-x   1 borchert sai         8600 Jan 16 10:52 /tmp/pubfile
theon$ chmod u+s /tmp/pubfile
theon$ ls -l /tmp/pubfile
-rwsrwxr-x   1 borchert sai         8600 Jan 16 10:52 /tmp/pubfile
theon$
```

- Das läßt sich aber ändern, wenn der Eigentümer des Programmes dem Programm das sogenannte „s-bit“ spendiert.
- Hierbei steht „s“ für **setuid**.
- Konkret bedeutet dies, dass das Programm mit den Privilegien des Programmeigentümers operiert und nicht mit denen des Aufrufers.

```
theon$ id
uid=819(gast) gid=207(guest)
theon$ /tmp/pubfile READ_ME
This is the file named READ_ME in the directory /home/borchert/pub.
theon$
```

- Nun klappt es für andere Benutzer.

- Wir haben nun den Fall, dass das Programm Privilegien besitzt, die der Aufrufer normalerweise nicht hat.
- Natürlich sollte so ein Programm nicht all seine Privilegien (im Beispiel die Rechte von *borchert*) dem Aufrufer preisgeben.
- Stattdessen hatte der Autor von *pubfile* die Absicht, dass nur die Dateien aus dem Unterverzeichnis *pub* der Öffentlichkeit zur Verfügung stehen sollen. Wenn es möglich ist, auf andere Dateien zuzugreifen oder gar beliebige Privilegien des Programmeigentümers ausnutzen zu können, dann würden Sicherheitslücken vorliegen.

# Die erste Lösung für *pubfile*



pubfile.c

```
/*
 * Display files within my pub directory.
 * Usage: pubfile {file}
 * WARNING: This program has several security flaws.
 * afb 2/2003
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>

const int BUFFER_SIZE = 8192;
const char* pubdir = "/home/borchert/pub";

int main(int argc, char** argv) {
   *argv++; --argc; /* skip command name */
   while (argc-- > 0) {
      /* ... process *argv++ ... */
   }
}
```

pubfile.c

```
/* process *argv++ */
char pathname[BUFFER_SIZE];
char buffer[BUFFER_SIZE];
int fd;
int count;

strcpy(pathname, pubdir);
strcat(pathname, "/");
strcat(pathname, *argv++);

if ((fd = open(pathname, O_RDONLY)) < 0) {
   perror(pathname); exit(1);
}
while ((count = read(fd, buffer, sizeof buffer)) > 0) {
   if (write(1, buffer, count) != count) {
      perror("write to stdout"); exit(1);
   }
}
if (count < 0) {
   perror(pathname); exit(1);
}
close(fd);
```

```
theon$ id
uid=819(gast) gid=207(guest)
theon$ /tmp/pubfile ../.ssh/id_rsa
-----BEGIN RSA PRIVATE KEY-----
[...]
-----END RSA PRIVATE KEY-----
theon$
```

- Unter Angabe eines relativen Pfadnamens können beliebige Dateien mit den Rechten des Benutzers *borchert* betrachtet werden.
- In diesem Beispiel wird der private RSA-Schlüssel ausgelesen, mit dessen Hilfe möglicherweise ein passwortloser Zugang auf andere Systeme mit den dortigen Privilegien von *borchert* eröffnet werden kann. Gelegentlich funktioniert das sogar auf dem gleichen System. Und hierfür genügte nur ein zu weitreichender Lesezugriff!

pubfile.c

```
    /* process *argv++ */
    char pathname[BUFFER_SIZE];
    /* ... */
    strcpy(pathname, pubdir);
    strcat(pathname, "/");
    strcat(pathname, *argv++);
```

- Hier wird der lokale Puffer *pathname* gefüllt, ohne auf die Größe des Puffers zu achten.
- Zwar mag *BUFFER_SIZE* großzügig gewählt sein, aber ein Argument auf der Kommandozeile kann deutlich länger sein.
- Die Frage ist ganz einfach: Was kann passieren, wenn der Indexbereich verlassen wird? Die Sprachdefinition von C selbst gibt keine Antwort darauf, abgesehen davon, dass das Verhalten dann als „undefiniert“ deklariert wird. Bei den gängigen Implementierungen mit einem rückwärts wachsenden Stack besteht die Möglichkeit, die Rücksprungadresse zu modifizieren und damit statt zum Aufrufer zu einem eingeschleusten Code springen zu lassen. Typischerweise kann der Code innerhalb des überlaufenden Puffers untergebracht werden.

- Ja. Aktuelles Beispiel: OP-TEE Trusted OS
- Zitat aus der Projektbeschreibung: „OP-TEE is a Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel running on Arm [..]“
- Das Buffer-Overrun-Problem bestand bei dem Systemaufruf *syscall_asymm_verify* (liegt der Funktion *TEE_AsymmetricVerifyDigest* der TEE Internal Core API zugrunde).
- Die Lücke wurde als CVE-2019-1010298 erfasst und hat den CVSS-Score 10.0 – das ist das Maximum, d.h. die Verletzung der sicheren bzw. vertrauenswürdigen Laufzeitumgebung (TEE) gelingt vollumfänglich und ohne vorherige Authentifizierung.

Siehe diesen Commit:

core/tee/tee_svc_cryp.c

```
TEE_Result syscall_asymm_verify(unsigned long state,
                        const struct utee_attribute *usr_params,
                        size_t num_params,
                        const void *data, size_t data_len,
                        const void *sig, size_t sig_len)
{
        /* ... */
        params = malloc(sizeof(TEE_Attribute) * num_params);
        if (!params)
                return TEE_ERROR_OUT_OF_MEMORY;
        res = copy_in_attrs(utc, usr_params, num_params, params);
        /* ... */
}
```

- Der Aufrufer kann *num_params* und *usr_params* beliebig preparieren. Insbesondere sind auch beliebig hohe Werte bei *num_params* möglich.
- Was passiert nun, wenn *num_params* so hoch gesetzt wird, dass es bei der Multiplikation zu einem Überlauf kommt?

- Contiki-NG: „The OS for Next Generation IoT Devices“.
- Zitat aus der Projektbeschreibung: „It focuses on dependable (secure and reliable) low-power communication and standard protocols, such as IPv6/6LoWPAN, 6TiSCH, RPL, and CoAP.“
- Das Buffer-Overrun-Problem bestand bei AQL-Abfragen (*Antelope Query Language*). Es genügt, wenn ein einzelnes lexikalisches Symbol (Token) mehr als 16 Zeichen umfasst.
- Die Lücke wurde als CVE-2018-1000804 erfasst und hat den CVSS-Score 10.0 – das ist das Maximum, d.h. der Einbruch gelingt vollumfänglich und dies gelingt über das Netzwerk ohne Voraussetzungen oder auch nur eine Authentifizierung. Beim *Internet of Things* (IoT) ist dies natürlich in besonderer Weise fatal, da die Software auf solchen Geräten häufig sich nur sehr schwer aktualisieren lässt – falls überhaupt.

Beschreibung übernommen von Issue #594:

db_options.h

```
#define DB_MAX_ELEMENT_SIZE 16
```

aql.h

```
typedef char value_t[DB_MAX_ELEMENT_SIZE];
```

aql-lexer.c

```
static int
next_token(lexer_t *lexer, const char *s)
{
  size_t length;

  length = strcspn(s, separators);
  /* ... */
  /* note: lexer->value is of type value_t */
  memcpy(lexer->value, s, length);
  (*lexer->value)[length] = '\0';
  return 1;
}
```

Und das ist kein uralter Code, sondern weniger als 10 Jahre alt.

- Das Heartbeat-Protokoll wurde in Ergänzung zum SSL-Protokoll definiert: RFC 6520
- Das Protokoll soll zwei Probleme lösen:
  - Eine schnellere Alternative zu *SO_KEEPALIVE*
  - Ein alternativer Ansatz zur *Path MTU Discovery*, nachdem die ursprünglich dafür gedachten ICMP-Pakete allzu häufig von Firewalls weggefiltert werden
- Im Rahmen des Protokolls können Pings geschickt werden mit Daten (Payload) und einer zufällig gewählten Ergänzung. Solche Pings werden dann beantwortet, wobei der Payload zusammen mit anderen zufälligen Daten zurückgeschickt wird.
- Der Payload hat eine variable Länge. Deswegen findet sich im Header eines Heartbeat-Pakets ein Feld mit zwei Bytes, das den Umfang der Payload-Daten spezifiziert.

ssl/ssl3.h

```
typedef struct ssl3_record_st
        {
/*r */  int type;               /* type of record */
/*rw*/  unsigned int length;    /* How many bytes available */
/*r */  unsigned int off;       /* read/write offset into 'buf' */
/*rw*/  unsigned char *data;    /* pointer to the record data */
/*rw*/  unsigned char *input;   /* where the decode bytes are */
/*r */  unsigned char *comp;    /* only used with decompression - malloc()ed *
/*r */  unsigned long epoch;    /* epoch number, needed by DTLS1 */
/*r */  unsigned char seq_num[8]; /* sequence number, needed by DTLS1 */
        } SSL3_RECORD;
```

- Eine typische Datenstruktur für einen Kommunikationspuffer, entnommen aus openssl-1.0.1f
- *data* zeigt auf (die bereits entschlüsselten) Daten, die wir über das Netzwerk erhalten haben.
- *length* gibt an, wieviele Bytes in *data* zum Lesen zur Verfügung stehen.

ssl/d1-both.c

```
        unsigned char *p = &s->s3->rrec.data[0], *pl;
        /* ... */
        /* Read type and payload length first */
        hbtype = *p++;
        n2s(p, payload);
        pl = p;
```

- *s−>s3−>rrec* ist vom Typ *SSL3_RECORD* und repräsentiert das eingelesene Datenpaket, in dem sich ein Heartbeat-Paket befindet.
- *p* zeigt auf den Anfang des Datenbereichs des eingelesenen Pakets.
- Dort ist zu Beginn der Typ des Heartbeat-Pakets (ein Byte) und der Umfang des beigefügten Payloads (zwei Bytes).
- *n2s* konvertiert zwei Bytes vom Netzwerk in *network byte order* in eine ganze Zahl (*short*).
- *payload* kann hier ein beliebiger Wert zwischen 0 und 65535 sein, der vollkommen frei von der anderen Seite gewählt werden kann.

# OpenSSL: Antwortpaket



ssl/d1-both.c

```
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,
           buffer, 3 + payload + padding);
```

- Hier wird ein Antwort-Paket geschnürt (in Reaktion zu einem Ping), bei der die erhaltene Payload zurückzuschicken ist mitsamt einer Ergänzung aus zufälligen Daten (*padding*).
- Mit Hilfe von *memcpy* wird von *pl* (zeigt an den Anfang der erhaltenen Payload) nach *bp* kopiert.
- Kopiert werden *payload* Bytes. Es wird nirgends überprüft, ob noch *payload* Bytes hinter *pl* belegt sind...

Kann ein Lesen (und Weitergeben) des Speicherinhalts jenseits des Eingabe-Puffers ein Problem darstellen?

- Ja! Ziemlich anschaulich erklärt es Randall Munroe in xkcd: `http://www.xkcd.com/1354/`
- Bruce Schneier dazu:

  *"Catastrophic" is the right word. On the scale of 1 to 10, this is an 11.*

In der Programmiersprache C hat es bereits erfolgreiche Einbrüche aufgrund folgender Programmierfehler gegeben:

- Unzureichende Überprüfung von Argumenten beim Eröffnen von Dateien, Ausführen von Kommandos oder anderen Systemaufrufen.
- Fehlende Einhaltung der Index-Grenzen eines Arrays. Gefahr besteht hier sowohl bei Arrays auf dem Stack als auch auf dem Heap (also per *malloc()* beschafft). Gefahr droht hier auch bei beliebten Funktionen der Bibliothek wie *strcpy*, *strcat*, *sprintf* und *gets*.
- Doppelte Freigabe eines Zeigers mit *free()*.
- Benutzung eines Zeigers, nachdem er bereits freigegeben worden ist.
- Weglassen des Formats bei *printf*. Statt *printf*(*s*) sollte besser *printf*("%s", *s*) verwendet werden.

- Leider ist die Vermeidung dieser Fehler nicht einfach.
- Selbst bei sicherheitsrelevanter Software wie der ssh (*secure shell*) oder der SSL-Bibliothek (*secure socket layer*) wurden immer wieder neue Fehler bei aufwendigen Untersuchungen des Programmtexts gefunden.
- Deswegen ist es bei C sinnvoll, bei systemnaher Software auf die Standard-Bibliotheken von C teilweise zu verzichten und stattdessen auf Alternativen auszuweichen, die die Verwendung sicherer Techniken unterstützen.

- Die Unterstützung dynamischer Zeichenketten in C ist nicht sehr ausgeprägt.
- Zwar ist es leicht möglich, mit *malloc()* ein Array der gewünschten Länge zu erhalten, aber danach gibt es keine zuverlässige Längeninformation mehr.
- *strlen* ist nur sinnvoll im Falle wohldefinierter Zeichenketten, da es nach dem Nullbyte sucht.
- Entsprechend haben Standardfunktionen wie *strcpy* oder *sprintf* keine Möglichkeit zu überprüfen, ob genügend Platz für das Ergebnis vorhanden ist.
- Folglich muß die Abschätzung dem Programmierer im Vorfeld überlassen werden, die dann häufig unterlassen wird oder fehlerhaft ist.

# Beispiel: Einlesen einer Zeile in eine Zeichenkette 350

readline.c

```
/*
 * Read a string of arbitrary length from a
 * given file pointer. LF is accepted as terminator.
 * 0 is returned in case of errors.
 * afb 3/2003
 */

#include <stdio.h>
#include <stdlib.h>

static const int INITIAL_LEN = 8;

char* readline(FILE* fp) {
   /* ... */
}
```

- Der Umgang mit Zeichenketten ist in C sehr umständlich, wenn die benötigte Länge nicht zu Beginn bekannt ist, wie dieses Beispiel demonstriert.

# Beispiel: Einlesen einer Zeile in eine Zeichenkette 351

readline.c

```
size_t len = 0; /* current length of string */
size_t alloc_len = INITIAL_LEN; /* allocated length */
char* buf = malloc(alloc_len);
int ch;

if (buf == 0) return 0;
while ((ch = getc(fp)) != EOF && ch != '\n') {
   if (len + 1 >= alloc_len) {
      alloc_len *= 2;
      char* newbuf = realloc(buf, alloc_len);
      if (newbuf == 0) {
         free(buf);
         return 0;
      }
      buf = newbuf;
   }
   buf[len++] = ch;
}
buf[len++] = '\0';
return realloc(buf, len);
```

Ein Ausweg besteht in der Schaffung einer alternativen Bibliothek für dynamische Zeichenketten in C, die folgende Anforderungen erfüllen sollte:

- Neben der eigentlichen Zeichenkette muß auch eine Längenangabe vorliegen.
- Bibliotheksfunktionen analog zu *strcpy()* und *strcat()* müssen unterstützt werden. Diese Funktionen müssen entweder die Längenangabe einhalten oder automatisch die Zeichenketten in ihrer Größe anpassen.
- Hinzu kommen Funktionen für die Initialisierung und die Freigabe von Zeichenketten.

Bei der Semantik gibt es zwei grundsätzliche Ansätze:

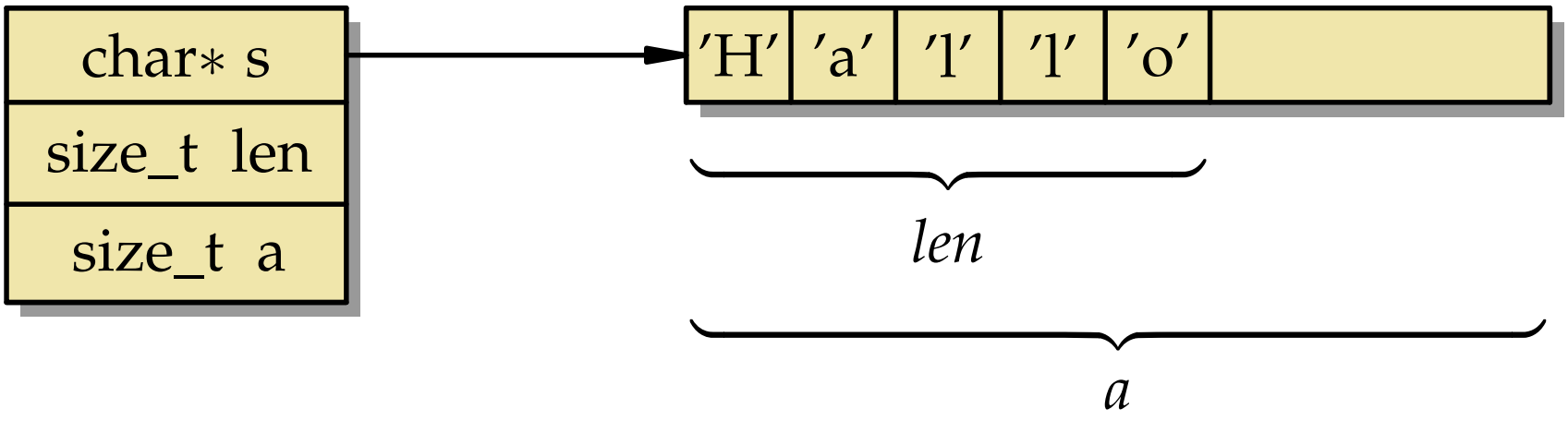
- Jede Zeichenkette ist in ihrer Repräsentierung unabhängig von allen anderen Zeichenketten und kann daher auch jederzeit frei verändert werden. Dies entspricht der traditionellen Vorgehensweise in C und der *string*-Template-Klasse in C++.
- Jede Zeichenkette ist konstant. Daher kann bei einer Operation analog zu *strcpy()* auf das Kopieren verzichtet werden. Änderungen erfordern hingegen das vorherige Anfertigen von Kopien. Dies entspricht der Vorgehensweise von Java. Hierfür wird entweder eine *garbage collection* oder der Einsatz von Referenzzählern benötigt.

- Eine C-Bibliothek, die dem ersten Ansatz folgt, wurde von Dan J. Bernstein entwickelt (u.a. für das Qmail-Paket).
- Später wurde sie von Felix von Leitner nachprogrammiert, um die Bibliothek unter der GPL (GNU General Public License) zur Verfügung stellen zu können.
- Zu finden ist sie unter `http://www.fefe.de/libowfat/`.

stralloc.h

```
typedef struct stralloc {
  char* s;
  size_t len;
  size_t a;
} stralloc;
```

- Diese öffentlich einsehbare Datenstruktur wird von Bernsteins Bibliothek verwendet.
- Der Zeiger *s* darf gleich 0 sein, um eine leere Zeichenkette zu repräsentieren. Dann müssen *len* und *a* ebenfalls 0 sein.




- *s* verweist auf einen Puffer der Länge *a*, in dem eine Zeichenkette der Länge *len* untergebracht ist. Es gilt: $len \leq a$.
- Im Gegensatz zu den normalen Zeichenketten unter C dürfen diese auch Nullbytes enthalten. Entsprechend gibt es keine Nullbyte-Terminierung.
- Bei Bedarf versuchen die *stralloc*-Funktionen automatisch, den Puffer (mit Hilfe von *realloc*) zu vergrößern (und geben **true** zurück, falls erfolgreich).

```
stralloc sa = {0};
```

- Wichtig ist die korrekte Initialisierung einer Variablen vom Typ *stralloc*. C sieht bei lokalen Variablen keine automatische Initialisierung vor, so dass hier die Initialisierung nicht vergessen werden darf.
- Damit wird übrigens nicht nur *sa.s* auf 0 initialisiert, sondern auch gleichzeitig *sa.len* und *sa.a* auf 0 gesetzt.

sareadline.c

```
/*
 * Read a string of arbitrary length from a
 * given file pointer. LF is accepted as terminator.
 * 1 is returned in case of success, 0 in case of errors.
 * afb 4/2003
 */

#include <stralloc.h>
#include <stdio.h>

int readline(FILE* fp, stralloc* sa) {
   sa->len = 0;
   for(;;) {
      if (!stralloc_readyplus(sa, 1)) return 0;
      if (fread(sa->s + sa->len, sizeof(char), 1, fp) <= 0) return 0;
      if (sa->s[sa->len] == '\n') break;
      ++sa->len;
   }
   return 1;
}
```

sareadline.c

```
int readline(FILE* fp, stralloc* sa) {
   sa->len = 0;
   /* ... */
}
```

- Hier wird zunächst die Länge auf 0 gesetzt.
- Es ist nicht unüblich, in einfachen Fällen direkt auf die *stralloc*-Datenstruktur zuzugreifen.
- Alternativ wäre es auch möglich gewesen mit *stralloc_copys*(*sa*, "") eine leere Zeichenkette zu kopieren.
- Generell dient *stralloc_copys* dazu, traditionelle nullbyte-terminierte Zeichenketten in C zu einem *stralloc*-Objekt zu kopieren.

sareadline.c

```
for(;;) {
   if (!stralloc_readyplus(sa, 1)) return 0;
   if (fread(sa->s + sa->len, sizeof(char), 1, fp) <= 0) return 0;
   if (sa->s[sa->len] == '\n') break;
   ++sa->len;
}
```

- Die **for**-Schleife behandelt das zeichenweise Einlesen, bis entweder das Zeilenende erkannt wird oder ein Fehler auftritt.
- Die Funktion *stralloc_readyplus* sorgt dafür, dass in *sa->s* mindestens ein Byte mehr Platz vorhanden ist, als die augenblickliche Länge *sa->len* beträgt.
- Nicht vergessen werden sollte die Überprüfung des Rückgabewerts. Bei 1 war die Operation erfolgreich, bei 0 konnte nicht genügend Speicher belegt werden.
- Wenn dies sichergestellt ist, kann mit *fread* das nächste Zeichen an der Position *sa->len* abgelegt werden.
- Wenn dies ein Zeilentrenner war, wird die **for**-Schleife beendet. Ansonsten wird das Zeichen akzeptiert, indem die Länge der Zeichenkette um 1 erhöht wird.

spubfile.c

```
while (argc-- > 0) {
   stralloc pathname = {0};
   char buffer[BUFFER_SIZE];
   int fd;
   int count;

   if (**argv == '.' || strchr(*argv, '/')) {
      fprintf(stderr, "invalid filename: %s\n", *argv);
      exit(1);
   }

   stralloc_copys(&pathname, pubdir);
   stralloc_cats(&pathname, "/");
   stralloc_cats(&pathname, *argv++);
   stralloc_0(&pathname);

   if ((fd = open(pathname.s, O_RDONLY)) < 0) {
      perror(pathname.s); exit(1);
   }
   /* ... copy contents of fd to stdout ... */
   close(fd);
}
```

spubfile.c

```
    stralloc_copys(&pathname, pubdir);
    stralloc_cats(&pathname, "/");
    stralloc_cats(&pathname, *argv++);
    stralloc_0(&pathname);
```

- Hinzugekommen ist hier die Funktion *stralloc_cats*, die eine traditionelle Zeichenkette an ein *stralloc*-Objekt anhängt.
- Die Funktion *stralloc_0* hängt genau ein Nullbyte an das *stralloc*-Objekt. Dies erlaubt es, *pathname.s* als traditionelle Zeichenkette in C zu verwenden — beispielsweise bei der Übergabe an die Funktion *open()*.
- Darüber hinaus wird in der korrigierten Version jeder Dateiname dahingehend überprüft, ob er mit einem Punkt beginnt, um sich insbesondere gegen die Verwendung von “.” und “..” zu schützen, und ob er einen Schrägstrich enthält, um sich gegen die Angabe relativer Pfadnamen zu schützen.

| | |
|---|---|
| *stralloc sa* = {0}; | Initialisierung einer Zeichenkette. |
| *stralloc_ready*(*sa*, *len*) | Bereitstellung von *len* Bytes. |
| *stralloc_readyplus*(*sa*, *len*) | Bereitstellung von *len* weiteren Bytes. |
| *stralloc_free*(*sa*) | Freigabe von *sa*. |
| *sa.s* | Direkter Zugriff auf den Zeiger. |
| *sa.len* | Länge der Zeichenkette. |
| *stralloc_copys*(*sa*, *s*) | Kopieren von *s* nach *sa*. |
| *stralloc_copy*(*sa1*, *sa2*) | Kopieren von *sa2* nach *sa1*. |
| *stralloc_cats*(*sa*, *s*) | Anhängen von *s* an *sa*. |
| *stralloc_cat*(*sa1*, *sa2*) | Anhängen von *sa2* an *sa1*. |
| *stralloc_0*(*sa*) | Anhängen eines Nullbytes an *sa*. |
| *stralloc_starts*(*sa*, *s*) | Findet sich *s* zu Beginn von *sa*? |
| *stralloc_diff*(*sa1*, *sa2*) | Vergleich analog zu *strcmp*. |
| *stralloc_diffs*(*sa*, *s*) | Vergleich analog zu *strcmp*. |

- *s* repräsentiert hier **char**∗, *sa* hingegen *stralloc*∗.

- Sicherheit sollte von Anfang an ein Kriterium sein. Es ist meistens ein hoffnungsloses Unterfangen, erst später Sicherheitsüberprüfungen einbauen zu wollen.
- Sicherheit sollte bei jedem Programm relevant sein, da sich sonst die Verwendung in einem sicherheitskritischen Kontext ausschließt. Nur bei temporären Wegwerf-Programmen können Sicherheitsbedenken wegfallen.
- Programme sollten nur ein Minimum an Privilegien erhalten. Häufig ist es ratsam, nicht nur auf `root`-Privilegien zu verzichten, sondern auch noch zusätzliche Restriktionen aufzunehmen wie die Limitierung des Ressourcen-Verbrauches und die Verwendung von `chroot`-Gefängnissen.
- Falls das Arbeiten mit Privilegien unverzichtbar ist, sollte das Aufteilen in mehrere Programme mit unterschiedlichen Privilegien in Betracht gezogen werden (*privilege separation*).

- Grundsätzlich sollte nichts und niemanden getraut werden, was von außen kommt.
- Bei der Überprüfung von Benutzereingaben sind Positivlisten (was ist erlaubt?) besser als Negativlisten (was ist gefährlich?).
- Sicherheit beruht auf Verantwortlichkeiten. Damit klar ist, welcher Programmteil für welche Überprüfungen verantwortlich ist, sollten entsprechende Vorgaben und Annahmen klar dokumentiert sein. So sollte beispielsweise innerhalb eines Programmes immer klar hervorgehen, wo mit ungeprüften Eingaben zu rechnen ist.
- Der wohldefinierte Bereich einer Programmiersprache sollte auf keinen Fall verlassen werden, unabhängig davon wie schwierig es sein mag, für Verletzungen passende Einbruchstechniken zu finden.

- Alle angebotenen automatischen Überprüfungen zur Übersetz- und Laufzeit sind zu verwenden.
- Wenn die Programmiersprache oder die Bibliothek nicht genügend automatische Überprüfungen mit sich bringen, ist es ratsam, Bibliotheken zu verwenden, die die Überprüfungen entweder durchführen oder überflüssig machen (Beispiel: **stralloc**-Bibliothek).
- Besser als das stille Abschneiden (Beispiel: *snprintf*()) ist die prinzipielle Unterstützung beliebig langer Eingaben. Der Speicherbedarf wird besser zentral limitiert als bei jeder einzelnen Eingabe.

- Die Grenzen aller Sicherheitsbemühungen sollten nicht vergessen werden.
- Das sicherste Programm nützt nichts, wenn die Bibliothek, der Compiler, das Betriebssystem oder die Hardware Sicherheitslücken aufweisen, die das Programm betreffen. Oder wenn sich über Nebeneffekte (wie etwa der Nutzung des L2-Caches) sicherheitsrelevante Inhalte erschließen lassen. Jüngste Beispiele sind Meltdown und Spectre.
- Ebenso ist der korrekte Umgang mit einer sicherheitskritischen Anwendung relevant. Das schwächste Glied in der Kette ist allzu häufig der Mensch (*social engineering*).