Coding Techniques

Andreas Zeller/Stephan Neuhaus

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Today's Specials

- KISS—Keep It Simple, Stupid
- Fail Safe and Fail Secure
- Locking Memory
- Sanitizing Memory
- Handles

Software Insecurity: Size

Software is *big*. Most interesting software systems consist of tens of thousands, hundreds of thousands or even millions of lines of code.

It is very difficult to make even small systems secure

Unforseen interactions between parts of the system can open you to security problems

Tenex Password Hack (1)

- Tenex was an operating system for the DEC-10
- password checking on files
- passwords were stored unencrypted
- password check only through (un-debuggable) system call

Tenex Password Hack (2)

- Tenex also had paged memory
- when a process accessed a page that was currently paged out to disk, a *page fault* occured
- feature that could notify a process whenever a page fault occured

Tenex Password Hack (3)

Here is the routine to check for the right password:

```
extern const char* lookup_password(const char* filename);
const int password_length = 14;

int password_equal(const char* a, const char* b) {
  int i;

  for (i = 0; i < 14; i++)
    if (a[i] != b[i])
      return 0;
  return 1;
}

int check (const char* filename, const char *given_password) {
  const char *actual_password = lookup_password(filename);
  return password_equal(actual_password, given_password);
}
```

How to Get Tenex Passwords to File f

1. Start with a 14-character password of ‘a’s. Set $i \leftarrow 0$ (The invariant is that we know i characters of the password.)

How to Get Tenex Passwords to File f

1. Start with a 14-character password of ‘a’s. Set $i \leftarrow 0$ (The invariant is that we know i characters of the password.)
2. Lay out the password such that the first $i + 1$ characters are in one memory page p , and the rest are in the adjacent page p'.

How to Get Tenex Passwords to File f

1. Start with a 14-character password of ‘a’s. Set $i \leftarrow 0$ (The invariant is that we know i characters of the password.)
2. Lay out the password such that the first $i + 1$ characters are in one memory page p , and the rest are in the adjacent page p'.
3. Force p' to be paged out.

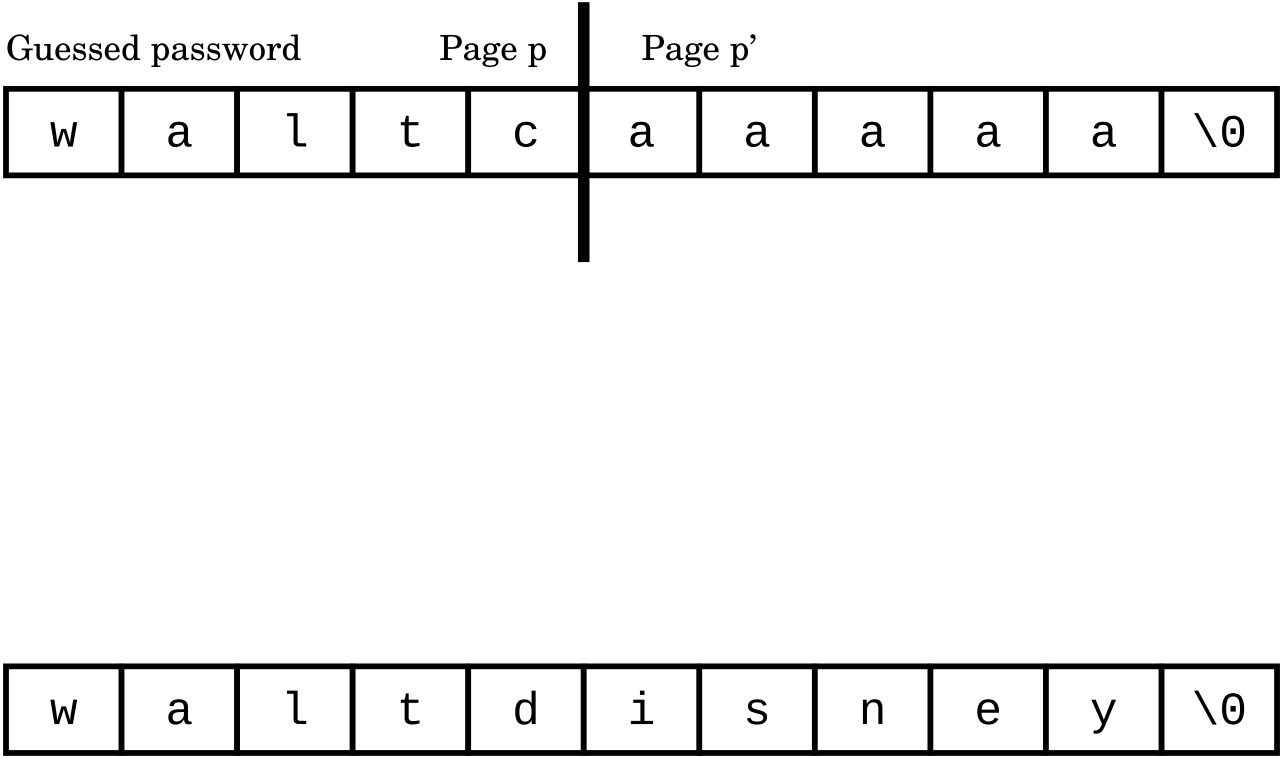
How to Get Tenex Passwords to File f

1. Start with a 14-character password of ‘a’s. Set $i \leftarrow 0$ (The invariant is that we know i characters of the password.)
2. Lay out the password such that the first $i + 1$ characters are in one memory page p , and the rest are in the adjacent page p'.
3. Force p' to be paged out.
4. Ask the operating system to open f with the password.

How to Get Tenex Passwords to File f

1. Start with a 14-character password of ‘a’s. Set $i \leftarrow 0$ (The invariant is that we know i characters of the password.)
2. Lay out the password such that the first $i+1$ characters are in one memory page p , and the rest are in the adjacent page p'.
3. Force p' to be paged out.
4. Ask the operating system to open f with the password.
5. If a positive answer comes back, you’re done. If a negative answer comes back, check if a page fault has occurred. If one has not occurred, increase the $i+1$-st character of the password by 1 and try again at step 3. If a page fault has occurred, increase i by 1 and try again at step 2.

Stealing Tenex Passwords



Guessed password

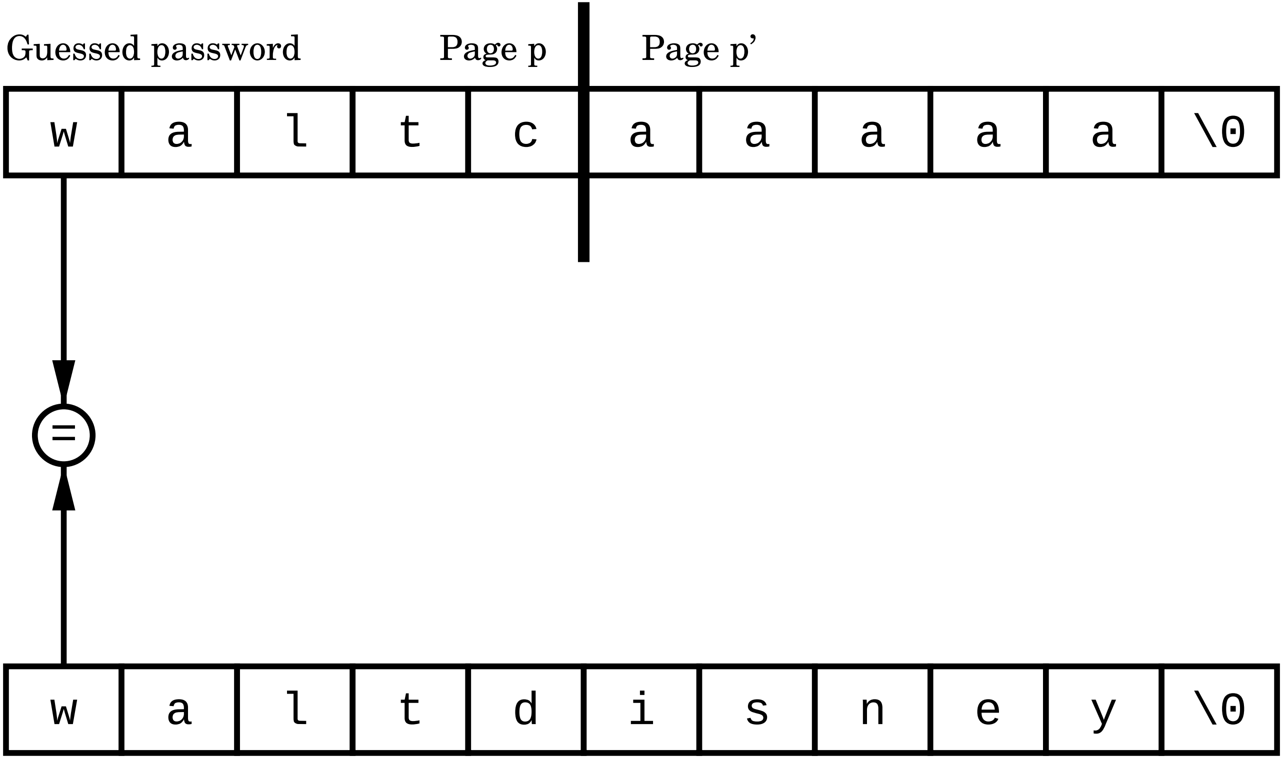
Page p | Page p’

w	a	l	t	c	a	a	a	a	a	\0

w	a	l	t	d	i	s	n	e	y	\0

Real password

Stealing Tenex Passwords



Guessed password

Page p

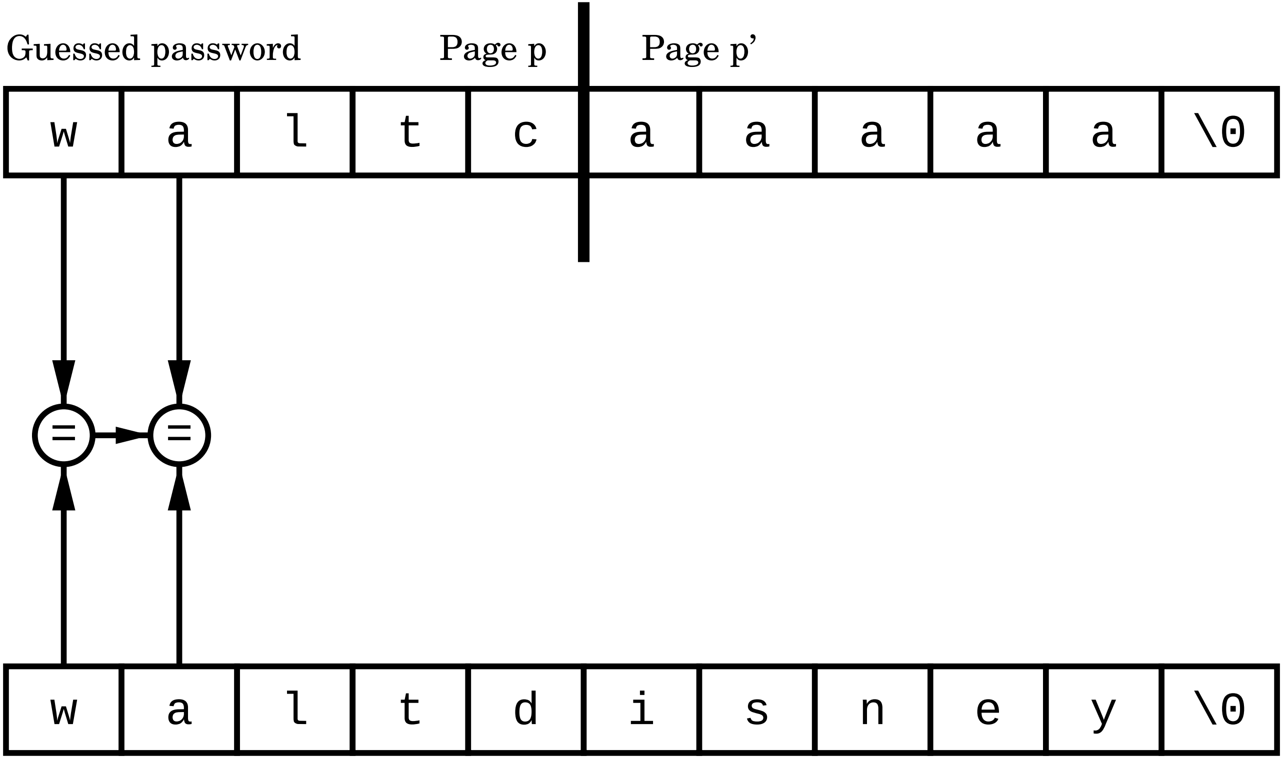
Page p’

w	a	l	t	c	a	a	a	a	a	\0

w	a	l	t	d	i	s	n	e	y	\0

Real password

Stealing Tenex Passwords



Guessed password

Page p

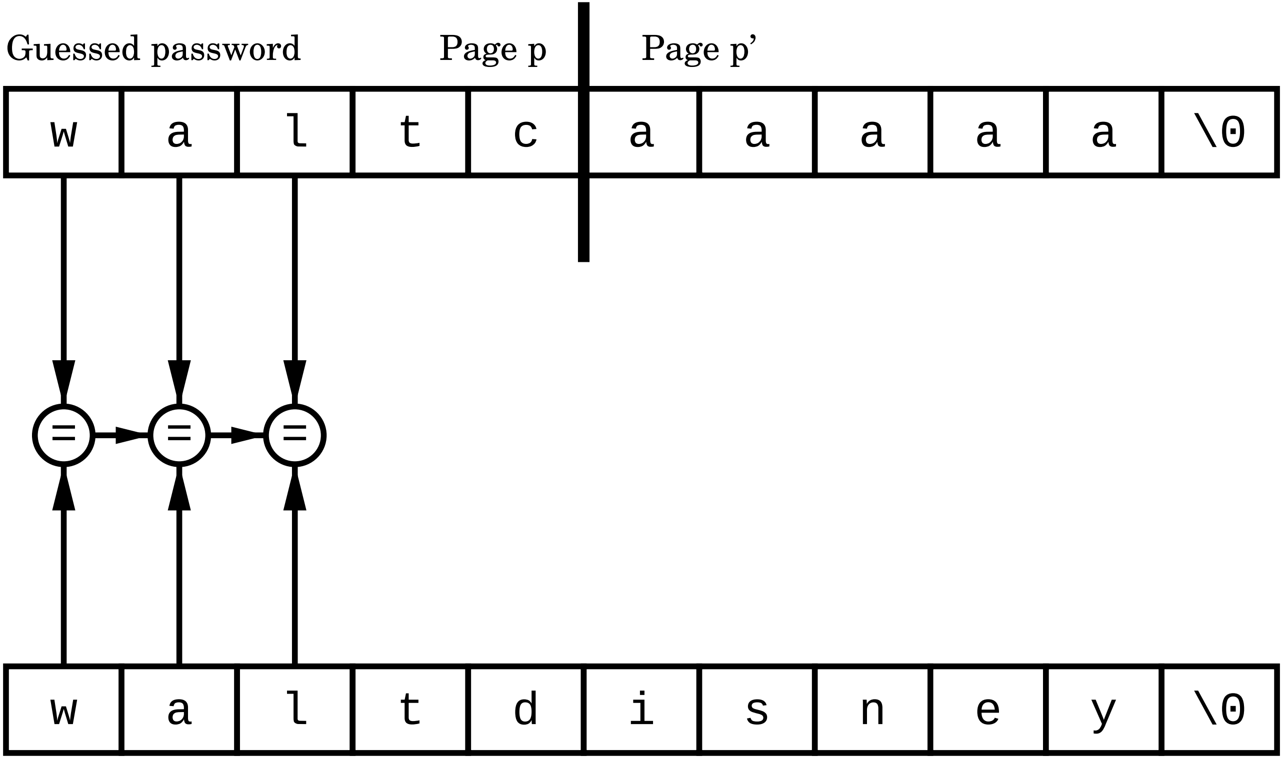
Page p’

w a l t c a a a a a \0

w a l t d i s n e y \0

Real password

Stealing Tenex Passwords



Guessed password

Page p

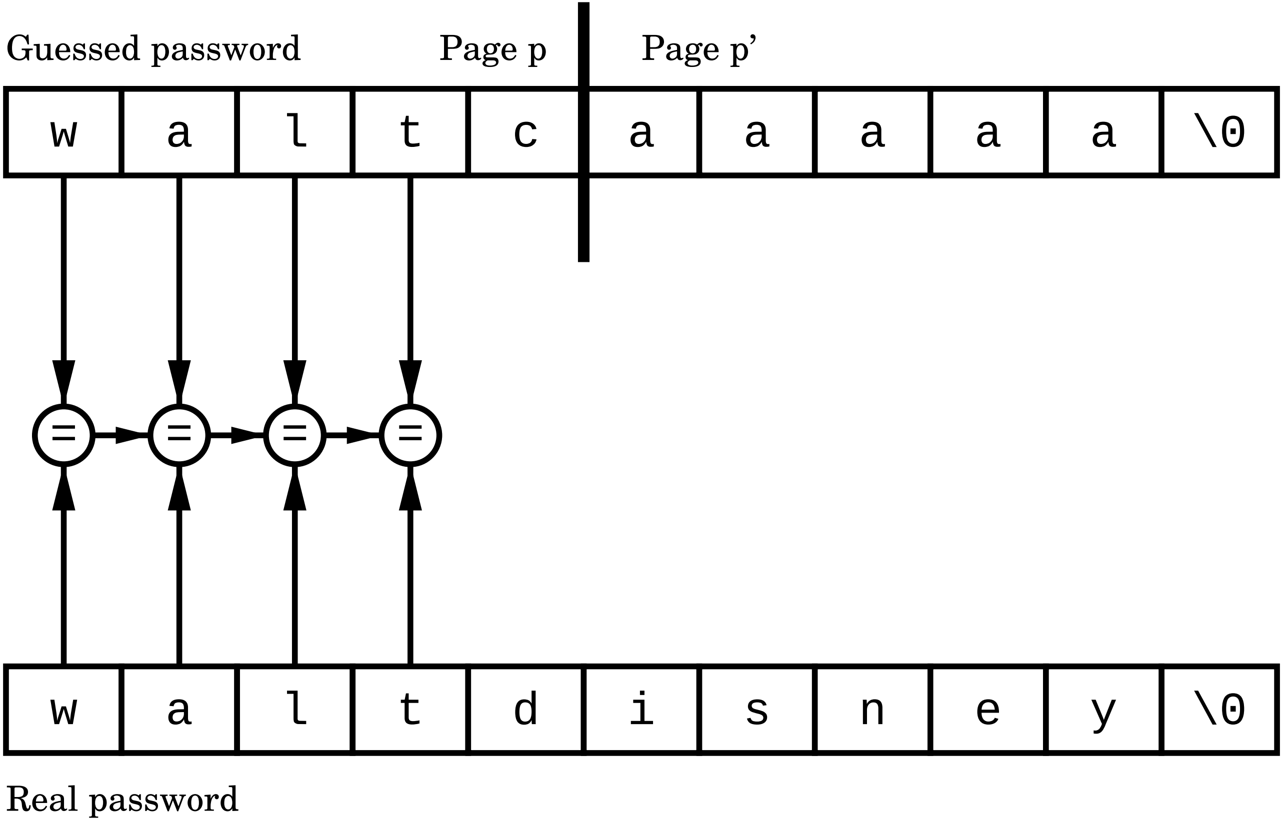
Page p’

w	a	l	t	c	a	a	a	a	a	\0

w	a	l	t	d	i	s	n	e	y	\0

Real password

Stealing Tenex Passwords



Guessed password

Page p

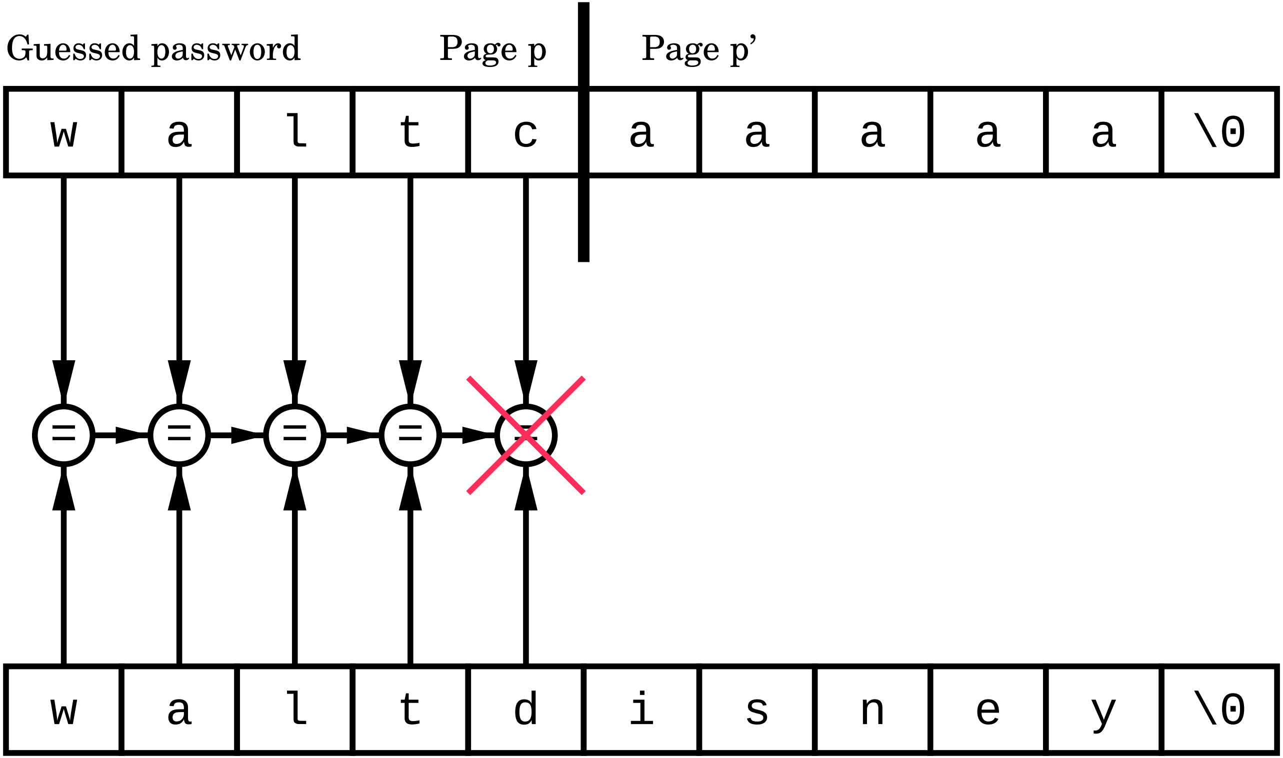
Page p’

w	a	l	t	c	a	a	a	a	a	\0

w	a	l	t	d	i	s	n	e	y	\0

Real password

Stealing Tenex Passwords



Guessed password

Page p

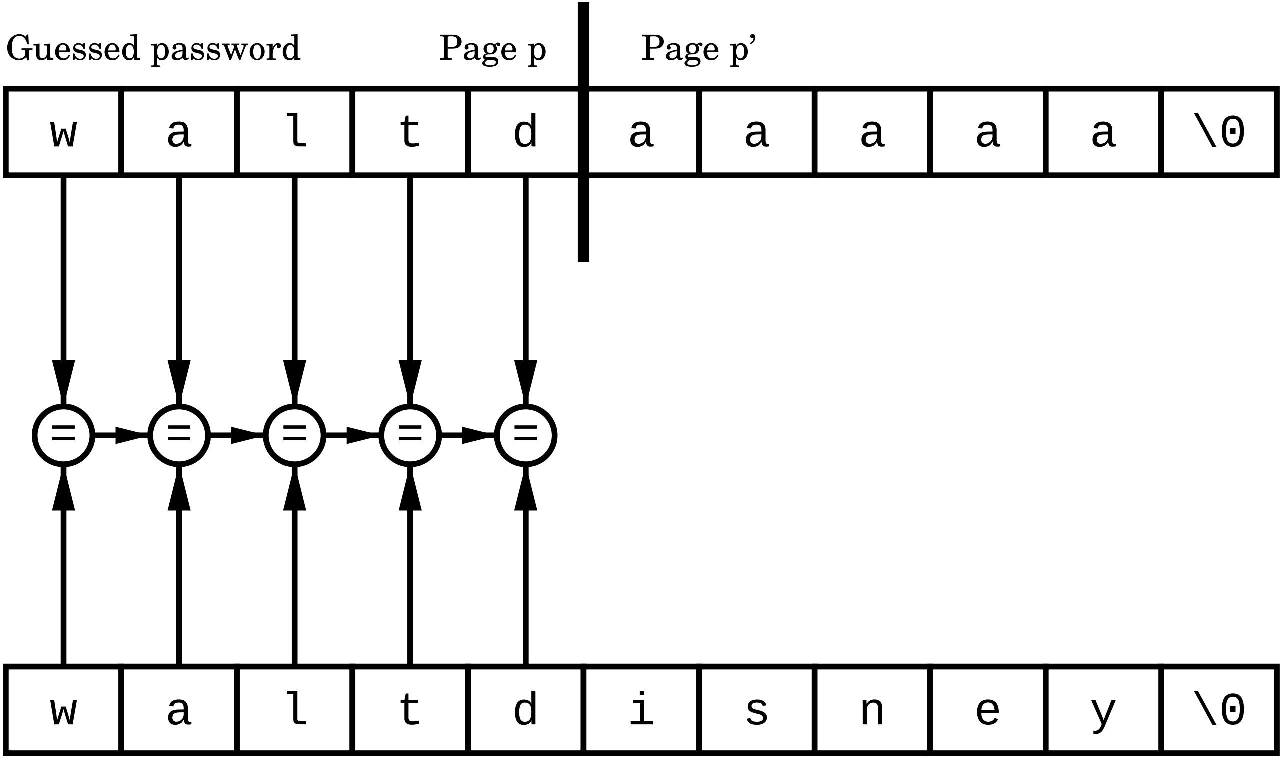
Page p’

w a l t c a a a a a \0

w a l t d i s n e y \0

Real password

Stealing Tenex Passwords



Guessed password

Page p

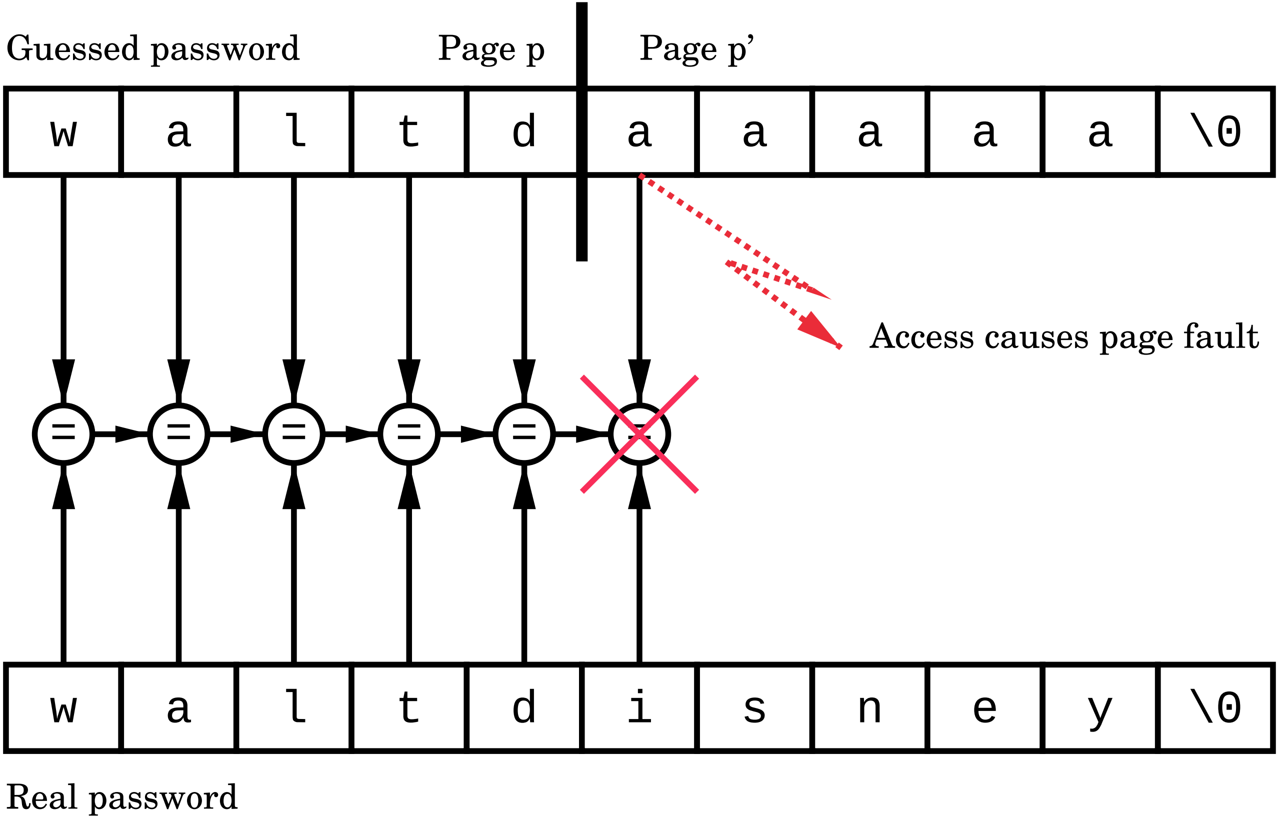
Page p’

w	a	l	t	d	a	a	a	a	a	\0

w	a	l	t	d	i	s	n	e	y	\0

Real password

Stealing Tenex Passwords



Software Complexity: Configurability

There was a time when Email addresses weren’t so simple as `user@mydomain.com`.

Software Complexity: Configurability

There was a time when Email addresses weren’t so simple as `user@mydomain.com`.

Email addresses could be ‘`user@mydomain.com`’ (internet mail address),

Software Complexity: Configurability

There was a time when Email addresses weren't so simple as `user@mydomain.com`.

Email addresses could be '`user@mydomain.com`' (internet mail address), '`minnehaha!kremvax!gorbachev`' ("bang path address"),

Software Complexity: Configurability

There was a time when Email addresses weren’t so simple as `user@mydomain.com`.

Email addresses could be ‘`user@mydomain.com`’ (internet mail address), ‘`minnehaha!kremvax!gorbachev`’ (“bang path address”), ‘`72223,10`’ (CompuServe address) etc.

Software Complexity: Configurability

There was a time when Email addresses weren't so simple as `user@mydomain.com`.

Email addresses could be '`user@mydomain.com`' (internet mail address), '`minnehaha!kremvax!gorbachev`' ("bang path address"), '`72223,10`' (CompuServe address) etc.

Email routing went via different paths (uucp dialup links, SMTP internet connections, etc.)

Software Complexity: Configurability

There was a time when Email addresses weren't so simple as `user@mydomain.com`.

Email addresses could be '`user@mydomain.com`' (internet mail address), '`minnehaha!kremvax!gorbachev`' ("bang path address"), '`72223,10`' (CompuServe address) etc.

Email routing went via different paths (uucp dialup links, SMTP internet connections, etc.)

The knight to slay this dragon of complexity was sendmail. Sendmail was configurable to handle all these addresses through a generalized string rewriting framweork.

Software Complexity: Configurability

There was a time when Email addresses weren’t so simple as `user@mydomain.com`.

Email addresses could be ‘`user@mydomain.com`’ (internet mail address), ‘`minnehaha!kremvax!gorbachev`’ (“bang path address”), ‘`72223,10`’ (CompuServe address) etc.

Email routing went via different paths (uucp dialup links, SMTP internet connections, etc.)

The knight to slay this dragon of complexity was sendmail. Sendmail was configurable to handle all these addresses through a generalized string rewriting framweork.

The downside of it was that no-one wanted to write or debug sendmail configuration files, because they were so complex.

Sendmail Example

```
S98
R$* < $m . >       $# local $@ $1      deliver mail to our domain
R$* < $m >         $# error $@ 4.5.1   can’t resolve domain?
R$* < $=w . $m . > $# local $@ $1      local hostname is OK
```

Who wants to write or debug this?

Sendmail Example

```
S98
R$* < $m . >       $# local $@ $1      deliver mail to our domain
R$* < $m >         $# error $@ 4.5.1   can’t resolve domain?
R$* < $=w . $m . > $# local $@ $1      local hostname is OK
```

Who wants to write or debug this?

```
# Towers of Hanoi
S49
RHANOI:$+       $:1 2 3$1
R$-$-$-$*[$+]   $:$1$2$3$4
R$-$-$-         $@$1$2$3
R$-$-$-@$*      $:$>49 $1$3$2$4
R$-$-$-$*       $:$>49 $2$3$1$4[Move Top Disk Of Peg $1 To Peg $3]
R$-$-$-$*       $:$3$2$1@$4
```

It’s Turing-complete, but it’s not nice! (Same holds for Intercal.)

What About It?

There was a time when this complexity was needed in order to deliver mail.

What About It?

There was a time when this complexity was needed in order to deliver mail.

This time is gone; email is now almost exclusively delivered via the Internet and SMTP ⇒ complexity isn’t needed anymore.

What About It?

There was a time when this complexity was needed in order to deliver mail.

This time is gone; email is now almost exclusively delivered via the Internet and SMTP ⇒ complexity isn't needed anymore.

New Mail Transfer Agents (MTAs) like `qmail` or `postfix` are not that excessively configurable.

What About It?

There was a time when this complexity was needed in order to deliver mail.

This time is gone; email is now almost exclusively delivered via the Internet and SMTP ⇒ complexity isn't needed anymore.

New Mail Transfer Agents (MTAs) like `qmail` or `postfix` are not that excessively configurable.

Also smaller (no a single monolithic I-can-do-it-all application.)

What About It?

There was a time when this complexity was needed in order to deliver mail.

This time is gone; email is now almost exclusively delivered via the Internet and SMTP ⇒ complexity isn't needed anymore.

New Mail Transfer Agents (MTAs) like `qmail` or `postfix` are not that excessively configurable.

Also smaller (no a single monolithic I-can-do-it-all application.)

Designed for security: `qmail` offers "qmail security guarantee": First person to detect a security flaw in qmail gets \$500. Offer open since March 1997, still no takers as of today; no `postfix` holes ever published.

Hints

- It’s easy to go overboard with configurability; the tendency is to make software as general as possible.

Hints

- It’s easy to go overboard with configurability; the tendency is to make software as general as possible.
- However, complexity is the natural enemy of security.

Hints

- It’s easy to go overboard with configurability; the tendency is to make software as general as possible.
- However, complexity is the natural enemy of security.
- Therefore, make your software just as configurable as it needs to be, but no more.

Hints

- It's easy to go overboard with configurability; the tendency is to make software as general as possible.
- However, complexity is the natural enemy of security.
- Therefore, make your software just as configurable as it needs to be, but no more.
- For example, in a MTA, you must make relaying an option, but not the format of `Date:` header lines (there is an RFC for that).

Hints

- It’s easy to go overboard with configurability; the tendency is to make software as general as possible.
- However, complexity is the natural enemy of security.
- Therefore, make your software just as configurable as it needs to be, but no more.
- For example, in a MTA, you must make relaying an option, but not the format of `Date:` header lines (there is an RFC for that).
- Also, beware of featuritis: Does a mail reader really need the capability to execute JavaScript or JScript code?

Hints

- It’s easy to go overboard with configurability; the tendency is to make software as general as possible.
- However, complexity is the natural enemy of security.
- Therefore, make your software just as configurable as it needs to be, but no more.
- For example, in a MTA, you must make relaying an option, but not the format of `Date:` header lines (there is an RFC for that).
- Also, beware of featuritis: Does a mail reader really need the capability to execute JavaScript or JScript code?
- (The configuration needs of a software can change over time.)

Fail Safe

To “fail safe” means that every failure must be checked for and must lead to a controlled reaction.

Fail Safe

To “fail safe” means that every failure must be checked for and must lead to a controlled reaction.

That means:

Fail Safe

To “fail safe” means that every failure must be checked for and must lead to a controlled reaction.

That means:

- Check *every* function call for errors.

Fail Safe

To “fail safe” means that every failure must be checked for and must lead to a controlled reaction.

That means:

- Check *every* function call for errors.
- Yes, even those that “cannot fail”.

Fail Safe

To “fail safe” means that every failure must be checked for and must lead to a controlled reaction.

That means:

- Check *every* function call for errors.
- Yes, even those that “cannot fail”.
- Do not remove this error checking code in production versions (usually “for performance reasons”).

Fail Safe

To “fail safe” means that every failure must be checked for and must lead to a controlled reaction.

That means:

- Check *every* function call for errors.
- Yes, even those that “cannot fail”.
- Do not remove this error checking code in production versions (usually “for performance reasons”).
- Make sure that you still can make a controlled exit if an error occurs: remember important variables etc.

Fail Safe

To “fail safe” means that every failure must be checked for and must lead to a controlled reaction.

That means:

- Check *every* function call for errors.
- Yes, even those that “cannot fail”.
- Do not remove this error checking code in production versions (usually “for performance reasons”).
- Make sure that you still can make a controlled exit if an error occurs: remember important variables etc.
- Always *explicitly* free all resources (memory, files, . . .) on termination; don’t rely on the operating system to do it for you.

Fail Secure

To “fail secure” means that failures must not lead to insecure behavior.

- If you encrypt data, don’t use fallback modes without encryption if something goes wrong.

Fail Secure

To “fail secure” means that failures must not lead to insecure behavior.

- If you encrypt data, don’t use fallback modes without encryption if something goes wrong.
- Don’t offer debugging modes with extended privileges and default passwords that are easily guessed.

Fail Secure

To “fail secure” means that failures must not lead to insecure behavior.

- If you encrypt data, don’t use fallback modes without encryption if something goes wrong.
- Don’t offer debugging modes with extended privileges and default passwords that are easily guessed.
- Offer secure defaults (probably the most overlooked recommendation of them all).

Fail Secure

To “fail secure” means that failures must not lead to insecure behavior.

- If you encrypt data, don’t use fallback modes without encryption if something goes wrong.
- Don’t offer debugging modes with extended privileges and default passwords that are easily guessed.
- Offer secure defaults (probably the most overlooked recommendation of them all).
- Don’t report success if success isn’t certain.

Reporting Success (1)

```
typedef enum { no_error, write_failed } write_error_t;

/* Writes a critical file. If this function returns with 0, the file
   has been written and committed to stable storage. Returns -1 on
   error. */
write_error_t write_critical_file() {
  FILE *fp = fopen("file", "w");
  const char* message = "This is a message";

  if (fwrite(message, strlen(message) + 1, 1, fp) != strlen(message) + 1) {
    error("Write failed"); return write_failed;
  }

  fclose(fp);
  return no_error;
}
```

Reporting Success (1)

```
typedef enum { no_error, write_failed } write_error_t;

/* Writes a critical file. If this function returns with 0, the file
   has been written and committed to stable storage. Returns -1 on
   error. */
write_error_t write_critical_file() {
  FILE *fp = fopen("file", "w");
  const char* message = "This is a message";

  if (fwrite(message, strlen(message) + 1, 1, fp) != strlen(message) + 1) {
    error("Write failed"); return write_failed;
  }

  fclose(fp);
  return no_error;
}
```

If the *fclose*(3) call fails, the file might not be on the disk!

Reporting Success (2)

```
typedef enum { no_error, write_failed, close_failed } write_error_t;

write_error_t write_critical_file() {
  FILE *fp = fopen("file", "w");
  const char* message = "This is a message";

  if (fwrite(message, strlen(message) + 1, 1, fp) != strlen(message) + 1) {
    error("Write failed"); return write_failed;
  }

  if (fclose(fp) != 0) { error("Close failed"); return close_failed; }

  return no_error;
}
```

Reporting Success (2)

```
typedef enum { no_error, write_failed, close_failed } write_error_t;

write_error_t write_critical_file() {
  FILE *fp = fopen("file", "w");
  const char* message = "This is a message";

  if (fwrite(message, strlen(message) + 1, 1, fp) != strlen(message) + 1) {
    error("Write failed"); return write_failed;
  }

  if (fclose(fp) != 0) { error("Close failed"); return close_failed; }

  return no_error;
}
```

File might be in kernel buffers even if *fclose*(3) succeeds.

Reporting Success (3)

```
typedef enum { no_error, write_failed, close_failed, sync_failed } write_error_t;

write_error_t write_critical_file() {
  FILE *fp = fopen("file", "w");
  const char* message = "This is a message";

  if (fwrite(message, strlen(message) + 1, 1, fp) != strlen(message) + 1) {
    error("Write failed"); return write_failed;
  }

  if (fclose(fp) != 0) { error("Close failed"); return close_failed; }

  if (fsync(fileno(fp)) != 0) { error("Sync failed"); return sync_failed; }

  return no_error;
}
```

Reporting Success (3)

```
typedef enum { no_error, write_failed, close_failed, sync_failed } write_error_t;

write_error_t write_critical_file() {
  FILE *fp = fopen("file", "w");
  const char* message = "This is a message";

  if (fwrite(message, strlen(message) + 1, 1, fp) != strlen(message) + 1) {
    error("Write failed"); return write_failed;
  }

  if (fclose(fp) != 0) { error("Close failed"); return close_failed; }

  if (fsync(fileno(fp)) != 0) { error("Sync failed"); return sync_failed; }

  return no_error;
}
```

Problem: invent a good strategy what to do if *fclose*(3) or *fsync*(3) fail. (Just reporting an error is often no good.)

A Little Puzzle

Check the following code:

```
unsigned char*
encrypt_file(const char* file_name, const char* user) {
  key_t* secret_key = lookup_secret_key(user);
  unsigned char* plaintext = read_file(file_name);
  unsigned char* ciphertext = encrypt(plaintext, secret_key);

  free(secret_key);
  free(plaintext);
  return ciphertext;
}
```

Looks good, doesn’t it?

A Little Puzzle

Check the following code:

```
unsigned char*
encrypt_file(const char* file_name, const char* user) {
  key_t* secret_key = lookup_secret_key(user);
  unsigned char* plaintext = read_file(file_name);
  unsigned char* ciphertext = encrypt(plaintext, secret_key);

  free(secret_key);
  free(plaintext);
  return ciphertext;
}
```

Looks good, doesn’t it?

It small, it has no buffer overflows or memory leaks...

A Little Puzzle

Check the following code:

```
unsigned char*
encrypt_file(const char* file_name, const char* user) {
  key_t* secret_key = lookup_secret_key(user);
  unsigned char* plaintext = read_file(file_name);
  unsigned char* ciphertext = encrypt(plaintext, secret_key);

  free(secret_key);
  free(plaintext);
  return ciphertext;
}
```

Looks good, doesn't it?

It small, it has no buffer overflows or memory leaks...

What could be wrong with it?

A Little Puzzle

Check the following code:

```
unsigned char*
encrypt_file(const char* file_name, const char* user) {
  key_t* secret_key = lookup_secret_key(user);
  unsigned char* plaintext = read_file(file_name);
  unsigned char* ciphertext = encrypt(plaintext, secret_key);

  free(secret_key);
  free(plaintext);
  return ciphertext;
}
```

Looks good, doesn't it?

It small, it has no buffer overflows or memory leaks...

What could be wrong with it?

Well, what *could* be wrong with it?

Handling Sensitive Data

A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!

Handling Sensitive Data

A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!

That's so important that we'll repeat it:

A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!

Handling Sensitive Data

A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!

That’s so important that we’ll repeat it:

A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!

Ways to accomplish this:

- Keep the key in memory, sanitize it afterwards

Handling Sensitive Data

A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!

That's so important that we'll repeat it:

A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!

Ways to accomplish this:

- Keep the key in memory, sanitize it afterwards
- Encrypt it before storing it

Keeping Data in Memeory

So the memory gets paged out to disk. What’s the deal?

Keeping Data in Memeory

So the memory gets paged out to disk. What’s the deal?

The deal is that

Keeping Data in Memeory

So the memory gets paged out to disk. What’s the deal?

The deal is that

- **A secret key (or any other sensitive piece of data)** ***must not ever*** **be outside your control in unencrypted form!** (thought I’d repeat that, just for good measure)

Keeping Data in Memeory

So the memory gets paged out to disk. What's the deal?

The deal is that

- **A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!** (thought I'd repeat that, just for good measure)
- Someone with access to the disk can get the secret

Keeping Data in Memeory

So the memory gets paged out to disk. What’s the deal?

The deal is that

- **A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!** (thought I’d repeat that, just for good measure)
- Someone with access to the disk can get the secret (and that’s *not* an academic threat!)

Keeping Data in Memeory

So the memory gets paged out to disk. What's the deal?

The deal is that

- **A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!** (thought I'd repeat that, just for good measure)
- Someone with access to the disk can get the secret (and that's *not* an academic threat!)
- Even if the data is officially erased (by overwriting it), it can still be restored (the phenomenon is called "remanence")

Keeping Data in Memeory

So the memory gets paged out to disk. What's the deal?

The deal is that

- **A secret key (or any other sensitive piece of data) *must not ever* be outside your control in unencrypted form!** (thought I'd repeat that, just for good measure)
- Someone with access to the disk can get the secret (and that's *not* an academic threat!)
- Even if the data is officially erased (by overwriting it), it can still be restored (the phenomenon is called "remanence")
- It's *very difficult* to erase data from a disk

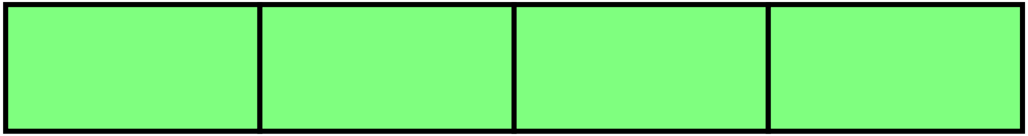
Keeping Data in Memory

```
#include <sys/mman.h>

/* Warning! mlock calls don’t stack! */
key_t* lookup_secret_key(const char* user) {
  key_t* ret = (key_t*) malloc(sizeof(key_t));
  if (ret != 0) {
    /* Must be root for this to succeed */
    if (mlock(ret, sizeof(ret)) == 0) {
      /* Proceed */
    } else {
      /* Handle error */
    }
  }
  return ret;
}

int release (const void* buf, size_t len) {
  free(buf);
  /* Must be root for this to succeed */
  return munlock(buf, len);
}
```

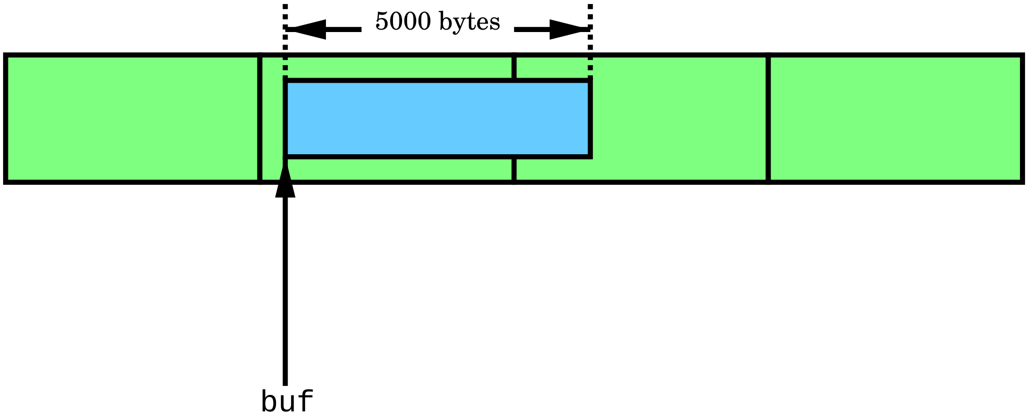
How mlock(2) works



🟩 Can be paged to disk

🟥 Locked in memory

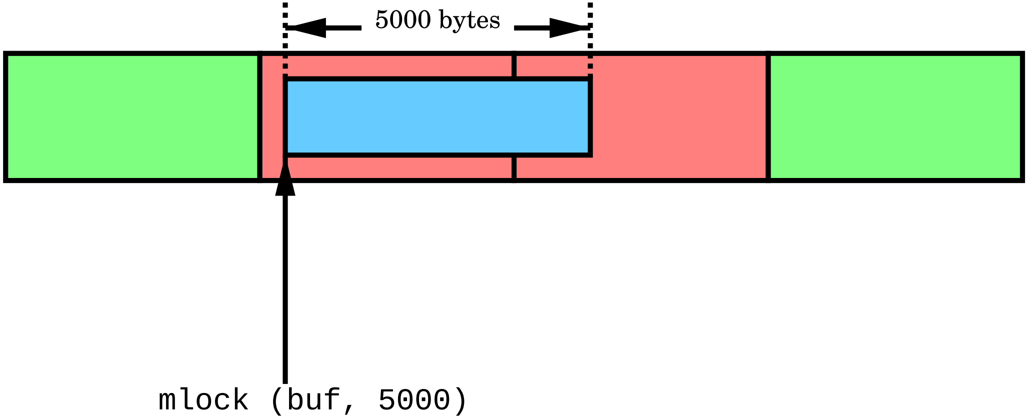
How mlock(2) works



🟩 Can be paged to disk

🟥 Locked in memory

How mlock(2) works

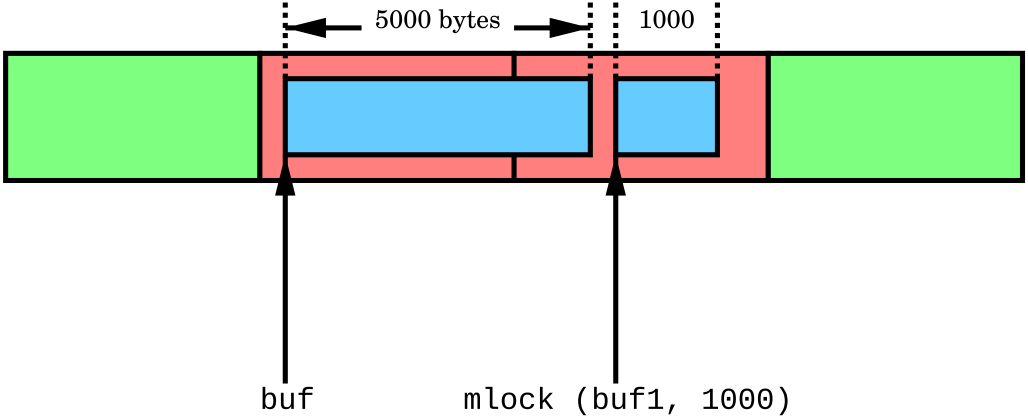


```
mlock (buf, 5000)
```

🟩 Can be paged to disk

🟥 Locked in memory

mlock(2) Calls Don’t Stack!



5000 bytes

1000

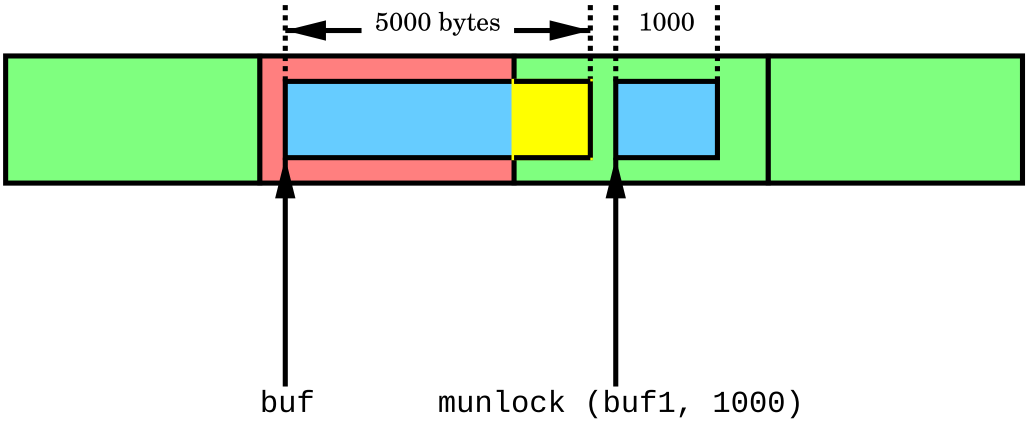
`buf`

`mlock (buf1, 1000)`

🟩 Can be paged to disk

🟥 Locked in memory

mlock(2) Calls Don’t Stack!



🟩 Can be paged to disk

🟥 Locked in memory

🟨 Shouldn’t be pageable, but is

mlock(2) Calls Don't Stack!

Solution:

- For every locked page, maintain a counter that says how many buffers are in that page.

mlock(2) Calls Don't Stack!

Solution:

- For every locked page, maintain a counter that says how many buffers are in that page.
- After freeing the memory allocated to a buffer, decrement the counter of each page the buffer is in by 1.

mlock(2) Calls Don’t Stack!

Solution:

- For every locked page, maintain a counter that says how many buffers are in that page.
- After freeing the memory allocated to a buffer, decrement the counter of each page the buffer is in by 1.
- When a counter reaches 0, there are no more buffers in the page, which can then be unlocked.

mlock(2) Calls Don't Stack!

Solution:

- For every locked page, maintain a counter that says how many buffers are in that page.
- After freeing the memory allocated to a buffer, decrement the counter of each page the buffer is in by 1.
- When a counter reaches 0, there are no more buffers in the page, which can then be unlocked.

It's probably easiest to combine that with the memory allocation functions.

Another Little Puzzle

```
unsigned char* encrypt_file(const char* file_name, const char* user) {
  key_t* secret_key = lookup_secret_key(user);     /* Uses mlock(2)! */
  unsigned char* plaintext = read_file(file_name); /* Uses mlock(2)! */
  unsigned char* ciphertext = encrypt(plaintext, secret_key);

  release(secret_key); release(plaintext);
  return ciphertext;
}

int release (const void* buf, size_t len) {
  free(buf);
  update_page_counts(buf, len);
  if (page_counts_are_zero(buf, len)) return munlock(buf, len); else return 0;
}
```

Looks good, doesn’t it?

Another Little Puzzle

```
unsigned char* encrypt_file(const char* file_name, const char* user) {
  key_t* secret_key = lookup_secret_key(user);    /* Uses mlock(2)! */
  unsigned char* plaintext = read_file(file_name); /* Uses mlock(2)! */
  unsigned char* ciphertext = encrypt(plaintext, secret_key);

  release(secret_key); release(plaintext);
  return ciphertext;
}

int release (const void* buf, size_t len) {
  free(buf);
  update_page_counts(buf, len);
  if (page_counts_are_zero(buf, len)) return munlock(buf, len); else return 0;
}
```

Looks good, doesn't it?

It small, uses *mlock*(2) . . .

Another Little Puzzle

```
unsigned char* encrypt_file(const char* file_name, const char* user) {
  key_t* secret_key = lookup_secret_key(user);    /* Uses mlock(2)! */
  unsigned char* plaintext = read_file(file_name); /* Uses mlock(2)! */
  unsigned char* ciphertext = encrypt(plaintext, secret_key);

  release(secret_key); release(plaintext);
  return ciphertext;
}

int release (const void* buf, size_t len) {
  free(buf);
  update_page_counts(buf, len);
  if (page_counts_are_zero(buf, len)) return munlock(buf, len); else return 0;
}
```

Looks good, doesn’t it?

It small, uses *mlock*(2) . . .

What *could* be wrong with it?

Another Little Puzzle

```
unsigned char* encrypt_file(const char* file_name, const char* user) {
  key_t* secret_key = lookup_secret_key(user);     /* Uses mlock(2)! */
  unsigned char* plaintext = read_file(file_name); /* Uses mlock(2)! */
  unsigned char* ciphertext = encrypt(plaintext, secret_key);

  release(secret_key); release(plaintext);
  return ciphertext;
}

int release (const void* buf, size_t len) {
  free(buf);
  update_page_counts(buf, len);
  if (page_counts_are_zero(buf, len)) return munlock(buf, len); else return 0;
}
```

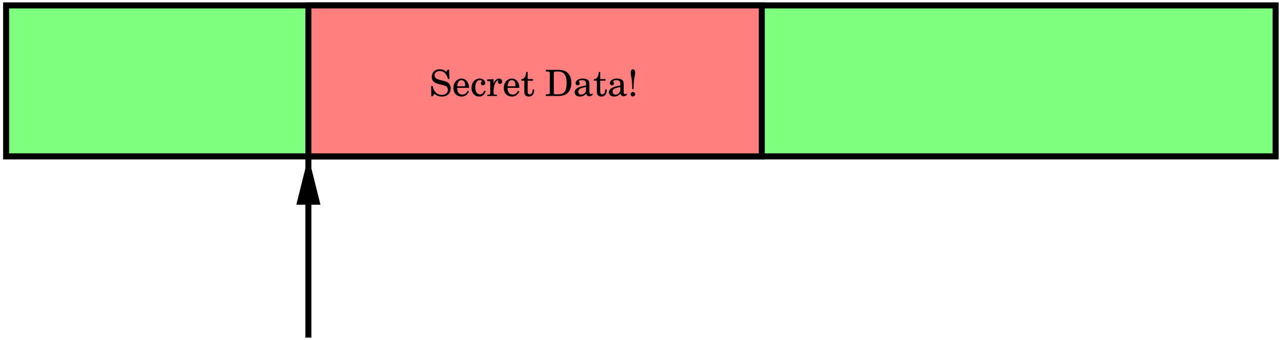
Looks good, doesn’t it?

It small, uses *mlock*(2) . . .

What *could* be wrong with it?

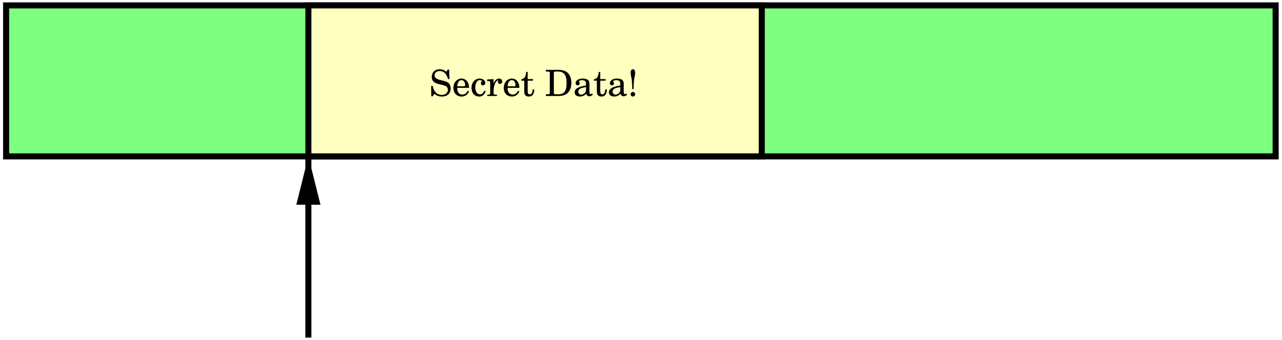
Allocating Memory

Allocating Memory



```
secret_buf = malloc(1000)
```

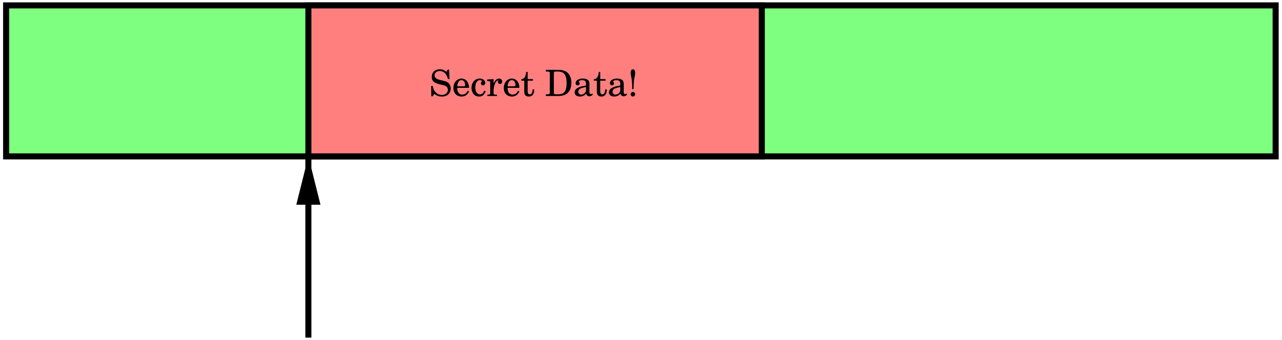
Allocating Memory



`free(secret_buf)`

Allocating Memory

Allocating Memory



`public_buf = malloc(1000)`

Public buffer now contains secret data!

Zeroing Memory

1. Allocate 1000 bytes

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material
3. Use key material, then release buffer

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material
3. Use key material, then release buffer
4. Allocate 1000 bytes for new buffer

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material
3. Use key material, then release buffer
4. Allocate 1000 bytes for new buffer
5. Fill only 500 bytes with harmless message

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material
3. Use key material, then release buffer
4. Allocate 1000 bytes for new buffer
5. Fill only 500 bytes with harmless message
6. Write 1000 bytes to file

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material
3. Use key material, then release buffer
4. Allocate 1000 bytes for new buffer
5. Fill only 500 bytes with harmless message
6. Write 1000 bytes to file
7. Release buffer

Result: 500 bytes of secret key material leaked

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material
3. Use key material, then release buffer
4. Allocate 1000 bytes for new buffer
5. Fill only 500 bytes with harmless message
6. Write 1000 bytes to file
7. Release buffer

Result: 500 bytes of secret key material leaked

Happens quickly: Difference between length and size of a buffer often not well understood.

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material
3. Use key material, then release buffer
4. Allocate 1000 bytes for new buffer
5. Fill only 500 bytes with harmless message
6. Write 1000 bytes to file
7. Release buffer

Result: 500 bytes of secret key material leaked

Happens quickly: Difference between length and size of a buffer often not well understood.

Better nip that problem in the bud!

So You Think It Can’t Happen?

Happened to Ethernet driver in Linux.

So You Think It Can't Happen?

Happened to Ethernet driver in Linux.

When a very small ICMP Echo (ping) packet was received, the return packet was incompletely initialized.

So You Think It Can’t Happen?

Happened to Ethernet driver in Linux.

When a very small ICMP Echo (ping) packet was received, the return packet was incompletely initialized.

Result: interesting information from the kernel’s memory was leaked.

So You Think It Can’t Happen?

Happened to Ethernet driver in Linux.

When a very small ICMP Echo (ping) packet was received, the return packet was incompletely initialized.

Result: interesting information from the kernel’s memory was leaked.

Could have been everything from Mom’s shopping list to passwords.

How to Avoid This Problem

```
#include <stdlib.h>

int release (void* buf, len_t len) {
  memset(buf, '\0', len);      /* <- Zeroize buffer before freeing */
  free(buf);

  update_page_counts(buf, len);
  if (page_counts_are_zero(buf, len))
    return munlock(buf, len);
  else
    return 0;
}
```

You must remember the size of the block you allocated. You can’t forget about that, like you can in “normal” C.

How to Avoid This Problem

```
#include <stdlib.h>

int release (void* buf, len_t len) {
  memset(buf, '\0', len);       /* <- Zeroize buffer before freeing */
  free(buf);

  update_page_counts(buf, len);
  if (page_counts_are_zero(buf, len))
    return munlock(buf, len);
  else
    return 0;
}
```

You must remember the size of the block you allocated. You can’t forget about that, like you can in “normal” C.

You *could* zeroize each block every time before freeing it, but chances are you’ll forget one of them ⇒ do it *once* in a library routine, then call that library routine.

Handles (1)

```
typedef unsigned char key_t[256];

unsigned char* encrypt_file(const char* plaintext, const char* user) {
  key_t secret_key = lookup_key(user); /* Not a pointer! */
  unsigned char ciphertext[512];

  /* ... */

  encrypt(plaintext, secret_key, ciphertext);
  fwrite(ciphertext, 1024, 1, fp); /* Oops! Writes secret key! */

  /* ... */
}
```

By mistake, the secret key gets written to disk.

Handles (1)

```
typedef unsigned char key_t[256];

unsigned char* encrypt_file(const char* plaintext, const char* user) {
  key_t secret_key = lookup_key(user); /* Not a pointer! */
  unsigned char ciphertext[512];

  /* ... */

  encrypt(plaintext, secret_key, ciphertext);
  fwrite(ciphertext, 1024, 1, fp); /* Oops! Writes secret key! */

  /* ... */
}
```

By mistake, the secret key gets written to disk.

The mistake is easy to spot in this example, but there are enough programs where these few lines are scattered among many files.

Handles (2)

```
typedef int key_handle_t;

unsigned char* encrypt_file(const char* plaintext, const char* user) {
  key_handle_t key_handle = lookup_key(user); unsigned char ciphertext[512];

  /* ... */
  encrypt(plaintext, key_handle, ciphertext);
  fwrite(ciphertext, 1024, 1, fp); /* Writes handle, not key */
  /* ... */
}
```

Using a handle instead of a pointer to the actual object makes it possible to check *every* use of the object.

Handles (2)

```
typedef int key_handle_t;

unsigned char* encrypt_file(const char* plaintext, const char* user) {
  key_handle_t key_handle = lookup_key(user); unsigned char ciphertext[512];

  /* ... */
  encrypt(plaintext, key_handle, ciphertext);
  fwrite(ciphertext, 1024, 1, fp); /* Writes handle, not key */
  /* ... */
}
```

10

Using a handle instead of a pointer to the actual object makes it possible to check *every* use of the object.

If you want to be paranoid, make handles difficult to guess. This will make accidental misuse of handles easy to detect.

Handles (2)

```
typedef int key_handle_t;

unsigned char* encrypt_file(const char* plaintext, const char* user) {
  key_handle_t key_handle = lookup_key(user); unsigned char ciphertext[512];

  /* ... */
  encrypt(plaintext, key_handle, ciphertext);
  fwrite(ciphertext, 1024, 1, fp); /* Writes handle, not key */
  /* ... */
}
```

10

Using a handle instead of a pointer to the actual object makes it possible to check *every* use of the object.

If you want to be paranoid, make handles difficult to guess. This will make accidental misuse of handles easy to detect.

For *extra* paranoia, you can code identifying information into the handle. ⇒ sharing of handles difficult between subjects.

Comparison Handles/Pointers

- Handles force separation of object implementation and use

Comparison Handles/Pointers

- Handles force separation of object implementation and use
- Handles awkward to use, not normal language objects

Comparison Handles/Pointers

- Handles force separation of object implementation and use
- Handles awkward to use, not normal language objects
- Handles need complete implementation, standard library has no generic support for handles

Comparison Handles/Pointers

- Handles force separation of object implementation and use
- Handles awkward to use, not normal language objects
- Handles need complete implementation, standard library has no generic support for handles
- Handles make access control easier (in many cases, it’s the only way to enable access controls at all)

Comparison Handles/Pointers

- Handles force separation of object implementation and use
- Handles awkward to use, not normal language objects
- Handles need complete implementation, standard library has no generic support for handles
- Handles make access control easier (in many cases, it's the only way to enable access controls at all)
- Handles reduce drastically the probability of accidentally leaking secret information

Coding for Testing

Most code that you write is difficult to test:

Coding for Testing

Most code that you write is difficult to test:

- Needs many other objects in order to test meaningfully.

Coding for Testing

Most code that you write is difficult to test:

- Needs many other objects in order to test meaningfully.
- Test cases are just too difficult to write.

Coding for Testing

Most code that you write is difficult to test:

- Needs many other objects in order to test meaningfully.
- Test cases are just too difficult to write.
- Time spent writing test cases is time not spent coding.

Coding for Testing

Most code that you write is difficult to test:

- Needs many other objects in order to test meaningfully.
- Test cases are just too difficult to write.
- Time spent writing test cases is time not spent coding.
- Besides, what could possibly go wrong?

Coding for Testing

Most code that you write is difficult to test:

- Needs many other objects in order to test meaningfully.
- Test cases are just too difficult to write.
- Time spent writing test cases is time not spent coding.
- Besides, what could possibly go wrong?

If you think like that, you may become famous...

Coding for Testing

Most code that you write is difficult to test:

- Needs many other objects in order to test meaningfully.
- Test cases are just too difficult to write.
- Time spent writing test cases is time not spent coding.
- Besides, what could possibly go wrong?

If you think like that, you may become famous... by being named in a CERT advisory as the programmer responsible for a security flaw!

Design Test Cases With the Code

- Use existing frameworks like CppUnit (for C++) or JUnit (for Java).

Design Test Cases With the Code

- Use existing frameworks like CppUnit (for C++) or JUnit (for Java).
- If you code in C, write your own framework

Design Test Cases With the Code

- Use existing frameworks like CppUnit (for C++) or JUnit (for Java).
- If you code in C, write your own framework, or contribute to CUnit (`http://cunit.sourceforge.net/`)

Design Test Cases With the Code

- Use existing frameworks like CppUnit (for C++) or JUnit (for Java).
- If you code in C, write your own framework, or contribute to CUnit (`http://cunit.sourceforge.net/`)
- Test extreme conditions (no input, empty input, one item, a million items, $\pm\infty$ for floats)

Design Test Cases With the Code

- Use existing frameworks like CppUnit (for C++) or JUnit (for Java).
- If you code in C, write your own framework, or contribute to CUnit (`http://cunit.sourceforge.net/`)
- Test extreme conditions (no input, empty input, one item, a million items, $\pm\infty$ for floats)
- Test fault conditions (bad password, wrong key, wrong encryption algorithm, ...)

Design Test Cases With the Code

- Use existing frameworks like CppUnit (for C++) or JUnit (for Java).
- If you code in C, write your own framework, or contribute to CUnit (`http://cunit.sourceforge.net/`)
- Test extreme conditions (no input, empty input, one item, a million items, $\pm\infty$ for floats)
- Test fault conditions (bad password, wrong key, wrong encryption algorithm, . . .)
- Test expected failures (decryption on modified ciphertext, signature verification on modified signature, negative values where only positive values are allowed, . . .)

Design Test Cases With the Code

- Use existing frameworks like CppUnit (for C++) or JUnit (for Java).
- If you code in C, write your own framework, or contribute to CUnit (`http://cunit.sourceforge.net/`)
- Test extreme conditions (no input, empty input, one item, a million items, $\pm\infty$ for floats)
- Test fault conditions (bad password, wrong key, wrong encryption algorithm, . . .)
- Test expected failures (decryption on modified ciphertext, signature verification on modified signature, negative values where only positive values are allowed, . . .)
- Test “impossible” conditions (bad parameters, enums outside the legal range, ‘NaN’s for floats. . .)

Learn from Testing (1)

If it is at all possible to provide your interface with “impossible” values, then the interface probably needs redesigning.

Learn from Testing (1)

If it is at all possible to provide your interface with “impossible” values, then the interface probably needs redesigning.

```
#include <string.h>

void* copy_memory(const void* block, int size) {
  void* ret = malloc(size);

  if (ret != 0)
    memcpy(ret, block, size);

  return ret;
}
```

Learn From Testing (2)

```
#include <string.h>

void* copy_memory(const void* block, size_t size) {
  void* ret = malloc(size);

  if (ret != 0) memcpy(ret, block, size);

  return ret;
}
```

Learn From Testing (2)

```
#include <string.h>

void* copy_memory(const void* block, size_t size) {
  void* ret = malloc(size);

  if (ret != 0) memcpy(ret, block, size);

  return ret;
}
```

- The block and its size clearly belong together.

Learn From Testing (2)

```
#include <string.h>

void* copy_memory(const void* block, size_t size) {
  void* ret = malloc(size);

  if (ret != 0) memcpy(ret, block, size);

  return ret;
}
```

- The block and its size clearly belong together.
- The block is an abstract data type that should only be accessed by handles.

Learn From Testing (2)

```
#include <string.h>

void* copy_memory(const void* block, size_t size) {
  void* ret = malloc(size);

  if (ret != 0) memcpy(ret, block, size);

  return ret;
}
```

- The block and its size clearly belong together.
- The block is an abstract data type that should only be accessed by handles.
- That way, all manipulations on blocks can be made locally, in one module (easier to verify).

Learn From Testing (2)

```
#include <string.h>

void* copy_memory(const void* block, size_t size) {
  void* ret = malloc(size);

  if (ret != 0) memcpy(ret, block, size);

  return ret;
}
```

- The block and its size clearly belong together.
- The block is an abstract data type that should only be accessed by handles.
- That way, all manipulations on blocks can be made locally, in one module (easier to verify).
- (That's really just common sense.)

Learn From Testing (3)

When it becomes impossible (or even very difficult) to provide an interface with “impossible” values, then it will be even more difficult for an attacker to inject “impossible” values through official channels.

Learn From Testing (3)

When it becomes impossible (or even very difficult) to provide an interface with “impossible” values, then it will be even more difficult for an attacker to inject “impossible” values through official channels.

The system has then become more secure *by design*.

Summary

- KISS—Keep It Simple, Stupid: Configurability, Size, Interconnections

Summary

- KISS—Keep It Simple, Stupid: Configurability, Size, Interconnections
- Fail Safe and Fail Secure

Summary

- KISS—Keep It Simple, Stupid: Configurability, Size, Interconnections
- Fail Safe and Fail Secure
- Locking Memory

Summary

- KISS—Keep It Simple, Stupid: Configurability, Size, Interconnections
- Fail Safe and Fail Secure
- Locking Memory
- Sanitizing Memory

Summary

- KISS—Keep It Simple, Stupid: Configurability, Size, Interconnections
- Fail Safe and Fail Secure
- Locking Memory
- Sanitizing Memory
- Handles

Summary

- KISS—Keep It Simple, Stupid: Configurability, Size, Interconnections
- Fail Safe and Fail Secure
- Locking Memory
- Sanitizing Memory
- Handles
- Testing

References

- Peter Gutmann, *Cryptographic Security Architecture*, Springer Verlag, 2003

References

- Peter Gutmann, *Cryptographic Security Architecture*, Springer Verlag, 2003
- Peter Gutmann, *Secure Deletion of Data from Magnetic and Solid-State Memory*, 1996 Usenix Security Symposion, `http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html`

References

- Peter Gutmann, *Cryptographic Security Architecture*, Springer Verlag, 2003
- Peter Gutmann, *Secure Deletion of Data from Magnetic and Solid-State Memory*, 1996 Usenix Security Symposion, `http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html`
- CUnit testing framework for C at `http://cunit.sourceforge.net/`