What is tamperproofing?

Ensure that a program executes as intended, even in the presence of an adversary who tries to disrupt, monitor, or change the execution.

- A tamperproofing algorithm
  - makes tampering difficult
  - detects when tampering has occurred
  - responds to the attack

What are typical attacks and defenses?

An attacker typically modifies the program with the intent to force it to chose a different execution path than the programmer intended:

1. remove code from and/or insert new code into the executable file prior to execution;
2. remove code from and/or insert new code into the running program;
3. affect the runtime behavior of the program through external agents such as emulators, debuggers, or a hostile operating system.

Algorithms

1. introspection, i.e. tamperproofed programs which monitor their own code to detect modifications.
2. various kinds of response mechanisms.
3. oblivious hashing algorithms which examine the state of the program for signs of tampering.
4. remote software authentication — determine that a program running on a remote machine has not been tampered with (WoW problem).

Outline

1. Definitions
2. Introspection
   - Checker Network
3. The Skype obfuscated protocol
4. Attacking self-hashing algorithms
5. State inspection
   - Overlapping instructions
6. Response Mechanisms
7. Discussion
How does the adversary attack \( P \)?

1. **Modify files:**
   - \( P \)'s executable file
   - Dynamic linker
   - Dynamic libraries

2. **Modify the operating system**

3. **Run \( P \) under emulation**

4. **Modify \( P \) while running under debugging**

---

What do we want?

Ensure \( P \) is healthy and the environment isn’t hostile:

1. Unadulterated **hardware** and **operating system**
2. Unmodified **\( P \)'s code**
3. Not running under **emulation**
4. Not being modified by a **debugger**
5. The right **dynamic libraries** have been loaded

---

Checking for tampering — code checking

- Check that \( P \)'s code hashes to a known value:

```c
if (hash(P's code) != 0xca7ca115) return false;
```
Checking for tampering — result checking

Instead of checking that the code is correct, CHECK can test that the result of a computation is correct.

```c
quickSort(A, n);
for (i = 0; i < (n - 1); i++)
    if (A[i] > A[i + 1])
        return false;
```

Checking for tampering — environment checking

- “Am I being run under emulation?”
- “Is there a debugger attached to my process?”
- “Is the operating system at the proper patch level?”

Environment checking — Checking for debugging

```c
#include <stdio.h>
#include <sys/ptrace.h>
int main() {
    if (ptrace(PTRACE_TRACEME))
        printf("I'm being traced!\n");
}
```

If you fail, you can assume you’ve been attached to a debugger:

```
gcc -g -o traced traced.c
gdb traced
(gdb) run
I'm being traced!
```
Environment checking — Checking for debugging

Here’s the output when first run normally and then under a debugger:

```c
> gcc -o cycles cycles.c
> cycles
elapsed 31528: Not debugged!
> gdb cycles
(gdb) handle SIGFPE noprint nostop
(gdb) run
elapsed 79272: Debugged!
```

How do we respond to tampering?

1. **Terminate** the program.
2. **Restore** the program to its correct state, by patching the tampered code.
3. Deliberately **return incorrect results**, maybe deteriorate slowly over time.
4. **Degrade** the performance of the program.
5. **Report the attack** for example by “phoning home”.
6. **Punish** the attacker by destroying the program or objects in its environment:
   - **DisplayEater** deletes your home directory.
   - Destroy the computer by repeatedly flashing the bootloader flash memory.
Checking by introspection

- Augment the program with functions that compute a hash over a code region to compare to an expected value.
- How can we be sure that the attacker won’t tamper with the hash computation itself?
  - build up a network of checkers and responders, so that checkers can check each other and responders can repair code that has been tampered with.
  - hide the hash values so they won’t give away the location of the checkers.
- We’ll see a clever attack on all introspection algorithms!
- ... And, We’ll see a clever counter-attack!

```
start = start_address;
end = end_address;
h = 0;
while (start < end) {
    h = h ⊕ start;
    start++;
}
if (h != expected_value)
    abort();
goto *h;
```

Attack model — Find the guards

1. Search for patterns in the static code, for example two code segment addresses followed by a test:
   ```
   start = 0xbabebabe;
   end = 0xca75ca75;
   while (start < end) {
   }
   ```
2. Search for patterns in the execution, such as data reads into the code.

Attack model — Disable the guards

1. Replace the if-statement by `if (0)...`:
   ```
   if (0)
       abort();
   ```
2. Pre-compute the hash value and substitute it into the response code:
   ```
   goto *expected_value
   ```
Tamperproofing Algorithm: Chang & Atallah

- Invented by two Purdue University researchers, Mike Atallah and Hoi Chang:

- Patented and with assistance from Purdue a start-up, Arxan, was spun off.

- Checkers compute a hash over a region and compare to the expected value.
- Checkers check the code and check each other as well!
- Build up a network of code regions: blocks of user code, checkers, and responders.
- When a tampered function is found repair it!
- Skype uses a similar technique.
- Multiple checkers can check the same region.
- Multiple responders can repair a tampered region.

```c
int main(int argc, char *argv[]) {
    int user_key = 0xca7ca115;
    int media[] = {10, 102};
    play(user_key,media,2);
}

int getkey(int user_key) {
    int player_key = 0xbabeca75;
    return user_key ^ player_key;
}

int decrypt(int user_key, int media) {
    int key = getkey(user_key);
    return media ^ key;
}

float decode(int digital) { return (float) digital; }
void play(int user_key, int media[], int len) {
    int i;
    for(i=0;i<len;i++)
        printf("%f\n",decode(decrypt(user_key,media[i])));
```
int main (int argc, char *argv[]) {
    A();
}
int A() {
    B();
}
int B() {
    ...
}

uint32 A_COPY[] = {0x83e58955, 0x72b820ec, 0xc7080486, ...};
uint32 B_COPY[] = {0x83e58955, 0xaeb820ec, 0xc7080486, ...};
int main (int argc, char *argv[]) {
    A_hash = hash(A);
    if (A_hash != 0x105AB23F)
        memcpy(A, A_COPY, 29*sizeof(uint32));
    A();
}
int A() {
    B_hash = hash(B);
    if (B_hash != 0x4f4205a5)
        memcpy(B, B_COPY);
    B();
}
int B() {
    ...
}

```c
int decrypt ( int user_key , int media ) {
    uint32 getkeyVal = hash((waddr_t)getkey,14);
    if (getkeyVal != 0xce1d400a)
        memcpy((waddr_t)getkey,getkeyCOPY,14*sizeof(uint32));
    int key = getkey(user_key);
    return media ^ key;
}

float decode ( int digital ) { return (float)digital; }

void play (int user_key , int media[], int len ) {
    if (decryptVal != 0x3764e45c)
        memcpy((waddr_t)decrypt,decryptCOPY,16*sizeof(uint32));
    int i;
    for (i =0; i<len; i++)
        printf("%f\n",decode(decrypt(user_key,media[i])));
}
```

Algorithm Chang & Atallah: Checker network

Introspection

Generating hash functions

- Prevent collusive attacks ⇒ generate a large number of different-looking hash functions.
- **Self-collusive** attacks = the adversary scans through the program for pieces of similar-looking code.
- No need to be “cryptographically secure”.
- No need to generate a uniform distribution of values.
- **Must be** simple, fast, stealthy!
typedef unsigned int uint32;
typedef uint32* addr_t;

uint32 hash1 (addr_t addr, int words) {
    uint32 h = *addr;
    int i;
    for (i = 1; i < words; i++) {
        addr ++;
        h ^= *addr;
    }
    return h;
}

uint32 hash2 (addr_t start, addr_t end) {
    uint32 h = *start;
    while (1) {
        start ++;
        if (start >= end) return h;
        h ^= *start;
    }
}

int32 hash3 (addr_t start, addr_t end, int step) {
    uint32 h = *start;
    while (1) {
        start += step;
        if (start >= end) return h;
        h ^= *start;
    }
}

uint32 hash4 (addr_t start, addr_t end, uint32 rnd) {
    addr_t t = (addr_t)((uint32)start + (uint32)end + rnd);
    uint32 h = 0;
    do {
        h += *((addr_t)(-(uint32)end-(uint32)rnd+(uint32)t));
        t++;
    } while (t < (addr_t)((uint32)end +
                        (uint32)end+(uint32)rnd));
    return h;
}

- Inline the function for better stealth.

- Will the compiler generate different code than for hash1???

- Step through the code region in more or less detail ⇒ balance performance and accuracy.

- Scan backwards.

- Obfuscate to prevent pattern-matching attacks: add (and then subtract out) a random value (rnd).
```c
uint32 hash5 (addr_t start, addr_t end, uint32 C) {
    uint32 h = 0;
    while (start < end) {
        h = C * (*start + h);
        start ++;
    }
    return h;
}
```

- Generate 2,916,864 variants, each less than 50 bytes of x86, by reordering basic blocks, inverting conditional branches, replacing multiplication instructions by combinations of shifts, adds, and address computations, permuting instructions within blocks, permuting register assignments, and replacing instructions with equivalents.

---

**The Skype obfuscated protocol**

- Voice-over-IP service where users are charged for computer-to-phone and phone-to-computer calls.
- The Skype client is heavily tamperproofed and obfuscated.
- 2005: Skype was bought by eBay for $2.6 billion.
- 2006: Hacked by two researchers at the EADS Corporate Research Center in France.

- The client binary contains:
  - hardcoded RSA keys
  - the IP address and port number of a known server
- Break the protection and build your own VoIP network!
Skype protection: Stage 1

1. Initialize
2. Load dlls
3. Erase
4. Decrypt
5. Load hidden dll table
6. ...

ENCRYPTED

ERASED

ERASED

pink: cleartext code, loads dlls.
blue: erase pink code, decrypts green code.
green: loads hidden dlls (yellow).
Erasing and hiding dlls: hard to recreate binary.

The Skype obfuscated protocol

Skype protection: Stage 2

- Check for debuggers:
  1. Signatures of known debuggers
  2. Timing tests

The Skype obfuscated protocol

Skype protection: Stage 3

- Checker network:
  1. Hash function computes the address of the next location to be executed!
  2. Hash functions are obfuscated, but not enough — attacked by pattern-matching.

The Skype obfuscated protocol

```c
uint32 hash7 () {
    addr_t addr;
    addr = (addr_t)((uint32)addr^((uint32)addr));
    addr = (addr_t)((uint32)addr + 0x688E5C);
    uint32 hash = 0x320E83 ^ 0x1C4C4;
    int bound = hash + 0xFFCC5AFD;
    do {
        uint32 data = *((addr_t)((uint32)addr + 0x10));
        goto b1; asm volatile(".byte 0x19");
        b1: hash = hash ⊕ data;
        addr -= 1; bound--;
    } while (bound!=0);
    goto b2;
    asm volatile(".byte 0x73");
    b2:
    goto b3;
    asm volatile(".word 0xC8528417,0xD8FBBD1,0xA36CFB2F");
    asm volatile(".word 0xE8D6E4B7,0xC0B8797A");
    asm volatile(".byte 0x61,0xBD");
    b3:
    hash-=0x4C49F346; return hash;
}
```

The Skype obfuscated protocol
Algorithm REWOS: Attacking self-hashing algorithms

Processors treat code and data differently.
TLBs (Translation Lookaside Buffers) and caches are split in separate parts for code and data.
In the hash-based algorithms code is accessed
1. as code (when it’s being executed) and
2. as data (when it’s being hashed).
⇒ sometimes a function will be read into the I-cache and sometimes into the D-cache.

Attack: modify the OS such that
1. redirect reads of the code to the original, unmodified program (hash values will be computed as expected!)
2. redirect execution of the code to the modified program (the modified code will get executed!)

How to attack introspection algorithms?
1. Analyze the code to locate the checkers, or
2. Analyze the code to locate the responders, then
3. Remove or disable them without destroying the rest of the program.
4. Attack can just as well be external to the program!
Algorithm REWOS: Attacking self-hashing algorithms

**Algorithm reWOS:** Attacking self-hashing algorithms

**ATTACK**(*P, K*):

1. Copy program *P* to *P*<sub>orig</sub>.
2. Modify *P* as desired to a hacked version *P*′.
3. Modify the operating system kernel *K* such that data reads are directed to *P*<sub>orig</sub>, instruction reads to *P*′.

**Typical memory management system:**

- On a **TLB miss** walk the page tables (slow), and update the TLB with the new virtual-to-physical address mapping.
- On the UltraSparc, the hardware gives the OS control on a TLB miss by **throwing one of two exceptions** depending on whether the miss was caused by a data or an instruction fetch.

**Example:**

- The attacker has modified the program to bypass a license-expired check.
- The original program pages are in blue.
- The modified program pages are in pink.
Algorithm REWOS: Attacking self-hashing algorithms

The program tries to read its own code in order to execute it
⇒ the processor throws an I-TLB-miss exception, the OS
updates the I-TLB to refer to the modified page.

The program tries to read its own code in order to hash the
processor throws a D-TLB-miss exception, and the OS
updates the D-TLB to refer to the original, unmodified, page.

What’s wrong with introspection algorithms?

- **Introspection algorithms**
  1. read their own code segment (unusual)!
  2. only check the validity of the code itself (not runtime data,
     function return values, ...).

- **Oblivious algorithms**
  1. detect tampering from the side-effects the code produces
  2. check the correctness of data and control-flow

Oblivious ⇒ the adversary should be unaware that his code is
being checked.

Outline

1. Definitions
2. Introspection
   - Checker Network
3. The Skype obfuscated protocol
4. Attacking self-hashing algorithms
5. State inspection
   - Overlapping instructions
   - Response Mechanisms
7. Discussion

Oblivious hashing

- More stealthy than introspection techniques.
  - We don’t read our own code!
- An advanced form of assertion checking:
  ```
  ASSERT x < 100;
  ASSERT y != null;
  ```
- Works on Java as well as binary code.
Challenging functions

- Adding assertion checks automatically is hard!
  - How can we know what values variables should have???
- Instead, call functions with challenge inputs:

```c
int challenge = 5;
int expected = 120;
int result = factorial(challenge);
if (result != expected)
    abort();
```

Be careful not to generate suspicious-looking hash values or challenge data:

```c
if (factorial(17) != 35687428096000)
    abort();
```

You can hide the hash value by making copies of every function:

```c
int challenge = 17;
if (factorial_orig(challenge) != factorial_copy(challenge))
    abort();
```

Algorithm TP.JJV: Oblivious hashing

- IDEA: overlap basic blocks of x86 instructions.
- When one block executes it also computes a hash over the second block!
- The hash is computed without reading the code!
- Invulnerable to memory splitting attacks!

Merge the blocks by interleaving the instructions, inserting jumps to maintain semantics:
The merged block has two entry points, $B_0$ and $B_1$.

Want the two blocks also to share instruction bytes.

Replace the jmp with xorl that takes a 4-byte literal argument:

$B_0$:

```
shll 2,%eax
xorl %ecx, next 4 bytes // used to be jmp $I_1$
```

$B_1$:

```
decl %eax
jmp $I_2$
nop
incl %eax
...`

The xorl instruction has, embedded in its immediate operand, the four bytes from decl;jmp;nop!

The Algorithm TPJJV: Oblivious hashing:

- Executing one block means also computing a hash over the other block into register %ecx!
- You can check the hash as usual.
- Clever use of the x86's architectural (mis-)features!
- Overhead: up to 3x slowdown.
**Outline**

1. Definitions
2. Introspection
   - Checker Network
3. The Skype obfuscated protocol
4. Attacking self-hashing algorithms
5. State inspection
   - Overlapping instructions
6. Response Mechanisms
7. Discussion

---

**Algorithm TPTEJ: Response Mechanisms**

```c
#include <time.h>
int global = 10;
...
if (time(0) % 2 == 0)
    printf("%i",10/global);
if (getpid() % 2 == 0)
    x = 5/global;
...
```

- CHECK checks for tampering,
- Later RESPOND takes action,
- Later still, the program actually fails

---

- RESPOND corrupts program state so that the actual failure follows much later
- Introduce a number of failure sites and probabilistically choose between them.
- Every time the attacker runs the hacked program it is likely to fail in one of the two green spots.
**spatial separation:** There should be as little static and dynamic connection between the RESPOND site and the failure site as possible.

**temporal separation:** A significant length of time should pass between the execution of RESPOND and the eventual failure.

**stealth:** The test, response, and failure code you insert in the program should be stealthy

**predictability:** Once the tamper response has been invoked, the program should eventually fail.

---

- Think about legal implications of your tamper response mechanism!
- Don’t deliberately destroy data...
- What if tamper-response was issued erroneously? (“I forgot my password, and after three tries the program destroyed my home directory!”)
- Watch out for unintended consequences. (the program crashes with a file open...)

---

- RESPOND to set a global pointer variable to NULL, causing the program to crash when the pointer is later dereferenced.
- If the program doesn’t have enough pointer variables TP·TCJ creates new ones by adding a layer of indirection to non-pointer variables.
- Assumes that there are enough global variables to choose from.
Algorithm TP TCJ: Example

1. Create a global pointer variable p_v.
2. To make the program crash you should set p_v to NULL. But where?
3. You want to avoid g and main since they will be on the call stack when f throws the pointer-reference-to-nil exception. (Check the stacktrace.)
4. Insert the failure-inducing code in h which is “many” calls away and not in the same call-chain as f.

Outline

1. Definitions
2. Introspection
   - Checker Network
3. The Skype obfuscated protocol
4. Attacking self-hashing algorithms
5. State inspection
   - Overlapping instructions
6. Response Mechanisms
7. Discussion

Trustworthiness

- Tamperproofing is about **trustworthiness**: Can I trust my program when it's running on an untrusted site?
- For us to trust P, the adversary
  - cannot add/remove/change P's code!
  - cannot modify P's environment!
- Essential for DRM, network gaming,...
Basic operations

- Check $P$’s environment:
  - Am I running under a debugger?
  - Am I running under emulation?
  - Has the OS been hacked?
- Check $P$’s code:
  - Have the executable bits been changed?
- Check $P$’s dynamic data:
  - Is $P$ in a legal executable state?

In practice...

- Use a combination of operations!
  - Check the environment
  - Check the code
  - Check the state
- You must check the checking code!
  - Simple attack: remove the checkers!
- The response must be stealthy!
  - Simple attack: trace back from failure!
- The detection must be stealthy!
  - Simple attack: detect reads of executable pages!