The ORCHIDS Intrusion Prevention System

Jean Goubault-Larrecq, Julien Olivain

ARCS Meeting — SFI, Nov. 2, 2006
Outline

1. Some Security Issues
2. Example: The `ptrace` Attack
3. Detecting The Attack, Using ORCHIDS
   - Demo
   - Under the Hood
4. The Architecture of ORCHIDS
5. Beyond: Additional Features, Further Attacks
6. Way Beyond
7. Conclusion
8. Other Things That Cannot Fit In The Talk
Some Types of Attacks

- Viruses, worms, trojans, buffer overflows, etc. (attacks on systems)
- Denial of service, IP/ARP spoofing, sniffing, etc. (network attacks)
- Attacks on html forms (perl, pgp), SQL insertion, IE/Word viruses [worms], etc. (attacks on applications)
Current Trends

- Systems and networks grow larger.
  More and more difficult to ensure any level of security.
Current Trends

- **Systems and networks grow larger.**
  More and more difficult to ensure any level of security.

- **Stakes are higher.**
  On-line data-bases [banking, health, taxes, ...],
  e-commerce, etc.
Current Trends

- Systems and networks grow larger. More and more difficult to ensure any level of security.

- Stakes are higher. On-line data-bases [banking, health, taxes, ...], e-commerce, etc.

- Attacks are more and more sophisticated, automated, and distributed. Ready-to-use packages [ask Google]
  Attacks requiring several steps, . . . each of them being innocuous in isolation
Current Trends

- Systems and networks grow **larger**.
  More and more difficult to ensure any level of security.

- Stakes are **higher**.
  On-line data-bases [banking, health, taxes, ...], e-commerce, etc.

- Attacks are more and more **sophisticated, automated, and distributed**.
  Ready-to-use packages [ask Google]
  Attacks requiring several steps,
  ... each of them being innocuous in isolation

- **New needs** in intrusion detection
  Tracking user configurations
  Detecting internal fraud
  Smartcards
Hey, Jean! You’re not just going to blabber along, right? Show them the ptrace attack for starters.
Hey, Jean! You’re not just going to blabber along, right? Show them the `ptrace` attack for starters.

Er, that’s what I meant to do… sure!

Me (confused)
1. Some Security Issues
2. Example: The `ptrace` Attack
3. Detecting The Attack, Using ORCHIDS
   - Demo
   - Under the Hood
4. The Architecture of ORCHIDS
5. Beyond: Additional Features, Further Attacks
6. Way Beyond
7. Conclusion
8. Other Things That Cannot Fit In The Talk
The ptrace Attack [Purczyński01,03]
The \texttt{ptrace} Attack [Purczyński01,03]

Exploits a problem on rights of spawned processes, with the \texttt{ptrace} system call.

Effect: a local, user-to-root attack.
The `ptrace` Attack [Purczyński01,03]

Exploits a problem on rights of spawned processes, with the `ptrace` system call.

**Effect:** a local, user-to-root attack.

- The `ptrace` call: used by all debuggers (benign in isolation!). Requires correlations.
- Subtle attack, based on a race condition in Linux 2.18 (Red Hat) kernels.
The Hacker’s View
The ptrace Attack [Purczyński01,03]

- Malicious program
  - pid = 100
  - `socket(AF_SECURITY, ...)` (unimplemented system call)
  - Not implemented
  - Search for a matching kernel module
  - Kernel mode

- User mode
  - pid = 101
  - modprobe
  - (kernel privileges)
  - not found

- Fail
  - `errno=ENOSYS`
The *ptrace* Attack [Purczyński01,03]

Malicious program

```
socket(AF_SECURITY, ...)  
(socket call, unimplemented system call)
```

Kernel mode

Not implemented

User mode

Search for a matching kernel module

```
modprobe  
(pid 101: root, root)
```

Kernel updates rights

```
pid=101  
(modprobe)
```

Fail

```
errno=ENOSYS  
```

```
not found
```

```
(pid 100)
```

```
Search for a matching kernel module
```

```
Not implemented
```

```
socket(AF_SECURITY, ...)
```

```
(socket call, unimplemented system call)
```

```
Fail
```

```
errno=ENOSYS
```
The `ptrace` Attack [Purczyński01,03]

- Malicious program (`pid = 100`)
  - `socket(AF_SECURITY, ...)` (unimplemented system call)
  - `ptrace(PTRACE_ATTACH, 101)`
    - `errno=ESRCH`
  - `ptrace(PTRACE_ATTACH, 101)`

- Kernel mode
  - Not implemented
    - Search for a matching kernel module
    - `pid=101`
      - `modprobe`
      - Kernel updates rights
        - `pid 101 : root, root`
      - Not found

- User mode
  - `errno=ENOSYS`
  - Fail

- Architecture
- Beyond
- Way Beyond
- Conclusion
- Misc
The `ptrace` Attack [Purczyński01,03]

Malicious program

Kernel mode

User mode

socket(AF_SECURITY, ...) (unimplemented system call)

Not implemented

Search for a matching kernel module

pid=101

modprobe ①

Insert shellcode

Kernel updates rights

pid 101 : root, root

(kernel privileges)

Shellcode runs with root privileges

exec ("/bin/sh")

Attacker has root privileges.

The `ptrace` Attack [Purczyński01,03]
Outline

1. Some Security Issues
2. Example: The `ptrace` Attack
3. Detecting The Attack, Using ORCHIDS
   - Demo
   - Under the Hood
4. The Architecture of ORCHIDS
5. Beyond: Additional Features, Further Attacks
6. Way Beyond
7. Conclusion
8. Other Things That Cannot Fit In The Talk
The Attack Signature

- We can count on the system logging important events. Here we count on the SNARE kernel module. We may also interface to the syslog facility.
The Attack Signature

- We can count on the system logging important events. Here we count on the SNARE kernel module. We may also interface to the syslog facility.

- ORCHIDS will now try to find patterns among these logged events: 

```
1. ptrace (ATTACH, Pid,Euid,Tgt )
2. exec ( Tgt )
3. ptrace (SYSCALL, Pid,Tgt )
4. ptrace (GETREGS, Pid,Tgt )
5. ptrace (POKETEXT, Pid,Tgt )
6. ptrace (DETACH, Pid,Tgt )
```
The Attack Signature

- We can count on the system logging important events. Here we count on the SNARE kernel module. We may also interface to the syslog facility.

- ORCHIDS will now try to find patterns among these logged events:

- Note that just detecting calls `ptrace` is not enough: this is used in everyday debugging activities, and is not indicative of an attack by itself.
Now let’s see Orchids in action.
Jean, did you show them that Orchids killed the offending user’s account? Did you explain them why?
Jean, did you show them that Orchids killed the offending user’s account? Did you explain them why?

Sure, Julien:

- The attacker may have left a backdoor in the system, allowing him to reenter at will. We should prevent him from using it later.

- Also, Orchids produces reports! Here, tracks attacker’s achievements.
Demo — Escaping Masking Attacks

Jean, have you shown them that Orchids was not fooled by *masking* attacks?
Jean, have you shown them that Orchids was not fooled by masking attacks?

Oh yes. The point is that the attacker may attempt to *drown* the intruder detection system under many similar events.

Goal: attempt to escape detection.

Let’s see how Orchids fares *under pressure*: let’s generate zillions of benign calls to `ptrace`. 
How Does ORCHIDS Detect It?

Imagine the following flow of events:

- open("/etc/passwd", "r", pid=58, euid=500)
- ptrace (ATTACH, pid=57, euid=500, 58)
- ptrace (ATTACH, pid=100, euid=500, 58)
- exec (prog="modprobe", pid=101)
- ptrace (ATTACH, pid=100, euid=500, 101)
- exit (pid=58)
- ptrace (SYSCALL, pid=100, 101)
- ptrace (GETREGS, pid=100, 101)
- ptrace (POKETEXT, pid=100, 101)
- ptrace (POKETEXT, pid=100, 101)
- ptrace (POKETEXT, pid=100, 101)
- ptrace (DETACH, pid=100, 101)

(What to come, in a nutshell: the ORCHIDS engine will basically simulate a form of alternating automata, with additional optimizations gotten by abstract interpretation of the formulae.)
Detecting the Attack

Flow of events:

- open("/etc/passwd", "r", pid=58, euid=500)
- ptrace(ATTACH, pid=57, euid=500, 58)
- ptrace(ATTACH, pid=100, euid=500, 101)
- exec(prog="modprobe", pid=101)
- ptrace(ATTACH, pid=100, euid=500, 101)
- exit(pid=58)
- ptrace(SYSCALL, pid=100, 101)
- ptrace(GETREGS, pid=100, 101)
- ptrace(POKETEXT, pid=100, 101)
- ptrace(POKETEXT, pid=100, 101)
- ptrace(POKETEXT, pid=100, 101)
- ptrace(DETACH, pid=100, 101)

Initially, ORCHIDS has no active thread.
Detecting the Attack

Flow of events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>&quot;/etc/passwd&quot;, &quot;r&quot;, pid=58, euid=500</td>
</tr>
<tr>
<td>ptrace</td>
<td>(ATTACH, pid=57, euid=500, 58)</td>
</tr>
<tr>
<td>ptrace</td>
<td>(ATTACH, pid=100, euid=500, 101)</td>
</tr>
<tr>
<td>exec</td>
<td>(prog=&quot;modprobe&quot;, pid=101)</td>
</tr>
<tr>
<td>ptrace</td>
<td>(ATTACH, pid=100, euid=500, 101)</td>
</tr>
<tr>
<td>exit</td>
<td>(pid=58)</td>
</tr>
<tr>
<td>ptrace</td>
<td>(SYSCALL, pid=100, 101)</td>
</tr>
<tr>
<td>ptrace</td>
<td>(GETREGS, pid=100, 101)</td>
</tr>
<tr>
<td>ptrace</td>
<td>(POKETEXT, pid=100, 101)</td>
</tr>
<tr>
<td>ptrace</td>
<td>(POKETEXT, pid=100, 101)</td>
</tr>
<tr>
<td>ptrace</td>
<td>(POKETEXT, pid=100, 101)</td>
</tr>
<tr>
<td>ptrace</td>
<td>(DETACH, pid=100, 101)</td>
</tr>
</tbody>
</table>

The signature contains no `open` event: skip.
Detecting the Attack

Flow of events:

open(”/etc/passwd”, ”r”, pid=58, euid=500)
trace (ATTACH, pid=57, euid=500, 58)
trace (ATTACH, pid=100, euid=500, 101)
exec (prog=”modprobe”, pid=101)
trace (ATTACH, pid=100, euid=500, 101)
exit (pid=58)

ptrace (SYSCALL, pid=100, 101)
ptrace (GETREGS, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (DETACH, pid=100, 101)
Detecting the Attack

Flow of events:

open("/etc/passwd", "r", pid=58, euid=500)
ptrace (ATTACH, pid=57, euid=500, 58)
ptrace (ATTACH, pid=100, euid=500, 101)
exec (prog="modprobe", pid=101)
ptrace (ATTACH, pid=100, euid=500, 101)
exit (pid=58)
ptrace (SYSCALL, pid=100, 101)
ptrace (GETREGS, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (DETACH, pid=100, 101)

1
2
3
4
5
6
7

Spawn thread: avoid masking attacks.
Detecting the Attack

Flow of events:

```
open("/etc/passwd", "r", pid=58, euid=500)
ptrace(ATTACH, pid=57, euid=500, 58)
ptrace(ATTACH, pid=100, euid=500, 101)
exec(prog="modprobe", pid=101)
ptrace(ATTACH, pid=100, euid=500, 101)
exit(pid=58)
ptrace(SYSCALL, pid=100, 101)
ptrace(GETREGS, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(DETACH, pid=100, 101)
```

```
1  ptrace (ATTACH, Pid,Euid,Tgt )
2  exec ( Tgt )
3  ptrace ( SYSCALL, Pid,Tgt )
4  ptrace ( GETREGS, Pid,Tgt )
5  (POKETEXT, Pid,Tgt )
6  ptrace ( DETACH, Pid,Tgt )
7
Pid = 57  Euid = 500  Tgt = 58
```

```
1  ptrace (ATTACH, Pid,Euid,Tgt )
2  exec ( Tgt )
3  ptrace ( SYSCALL, Pid,Tgt )
4  ptrace ( GETREGS, Pid,Tgt )
5  (POKETEXT, Pid,Tgt )
6  ptrace ( DETACH, Pid,Tgt )
7
Pid = 100  Euid = 500  Tgt = 101
```

No need to spawn thread: would violate shortest runs.
Detecting the Attack

Flow of events:

- open("/etc/passwd", "r", pid=58, euid=500)
- ptrace(ATTACH, pid=57, euid=500, 58)
- ptrace(ATTACH, pid=100, euid=500, 101)
- exec(prog="modprobe", pid=101)
- ptrace(ATTACH, pid=100, euid=500, 101)
- exit(pid=58)
- ptrace(SYSCALL, pid=100, 101)
- ptrace(GETREGS, pid=100, 101)
- ptrace(POKETEXT, pid=100, 101)
- ptrace(POKETEXT, pid=100, 101)
- ptrace(DETACH, pid=100, 101)
Detecting the Attack

Flow of events:

open("/etc/passwd", "r", pid=58, euid=500)
ptrace (ATTACH, pid=57, euid=500, 58)
ptrace (ATTACH, pid=100, euid=500, 101)
exec (prog="modprobe", pid=101)
ptrace (ATTACH, pid=100, euid=500, 101)
exit (pid=58)
Detecting the Attack

Flow of events:

```
open("/etc/passwd", "r", pid=58, euid=500)
ptrace (ATTACH, pid=57, euid=500, 58)
ptrace (ATTACH, pid=100, euid=500, 101)
exec (prog="modprobe", pid=101)
ptrace (ATTACH, pid=100, euid=500, 101)
exit (pid=58)
```
Detecting the Attack

Flow of events:

```plaintext
open("/etc/passwd", "r", pid=58, euid=500)
ptrace(ATTACH, pid=57, euid=500, 58)
ptrace(ATTACH, pid=100, euid=500, 101)
exec(prog="modprobe", pid=101)
ptrace(ATTACH, pid=100, euid=500, 101)
exit(pid=58)
```

```
ptrace(SYSCALL, pid=100, 101)
ptrace(GETREGS, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(DETACH, pid=100, 101)
```

---

Keep run
ATTACH-exec-SYSCALL-GETREGS for reporting.
Detecting the Attack

Flow of events:

open("/etc/passwd", "r", pid=58, euid=500)
ptrace(ATTACH, pid=57, euid=500, 58)
ptrace(ATTACH, pid=100, euid=500, 101)
exec(prog="modprobe", pid=101)
ptrace(ATTACH, pid=100, euid=500, 101)
ptrace(DETACH, pid=100, 101)
exit(pid=58)
ptrace(SYSCALL, pid=100, 101)
ptrace(GETREGS, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(DETACH, pid=100, 101)

Do not spawn thread (shortest runs again) (prefer A– over -A-, –A on trace AAA). Formally: keep lexicographically smallest sequences of event numbers.
Detecting the Attack

Flow of events:

```plaintext
open("/etc/passwd", "r", pid=58, euid=500)
ptrace(ATTACH, pid=57, euid=500, 58)
ptrace(ATTACH, pid=100, euid=500, 101)
exec(prog="modprobe", pid=101)
ptrace(ATTACH, pid=100, euid=500, 101)
exit(pid=58)
```

Do not spawn thread (shortest runs again) (prefer A– over -A-, –A on trace AAA). Formally: keep lexicographically smallest sequences of event numbers.
Detecting the Attack

Flow of events:

open("/etc/passwd", "r", pid=58, euid=500)
ptrace(ATTACH, pid=57, euid=500, 58)
ptrace(ATTACH, pid=100, euid=500, 101)
exec(prog="modprobe", pid=101)
ptrace(ATTACH, pid=100, euid=500, 101)
ptrace(DETACH, pid=100, 101)
ptrace(SYSCALL, pid=100, 101)
ptrace(GETREGS, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(POKETEXT, pid=100, 101)
ptrace(DETACH, pid=100, 101)

Do not spawn thread (shortest runs again) (prefer A– over -A-, –A on trace AAA).
Formally: keep lexicographically smallest sequences of event numbers.
Detecting the Attack

Flow of events:

- `open("/etc/passwd", "r", pid=58, euid=500)`
- `ptrace(ATTACH, pid=57, euid=500, 58)`
- `ptrace(ATTACH, pid=100, euid=500, 101)`
- `exec(prog="modprobe", pid=101)`
- `ptrace(ATTACH, pid=100, euid=500, 101)`
- `exit(pid=58)`
- `ptrace(SYSCALL, pid=100, 101)`
- `ptrace(GETREGS, pid=100, 101)`
- `ptrace(POKETEXT, pid=100, 101)`
- `ptrace(POKETEXT, pid=100, 101)`
- `ptrace(POKETEXT, pid=100, 101)`
- `ptrace(DETACH, pid=100, 101)`

```
1 ptrace (ATTACH, Pid,Euid,Tgt ) exec (Tgt ) 3 ptrace (SYSCALL, Pid,Tgt )

2 ptrace (ATTACH, Pid,Euid,Tgt ) exec (Tgt ) 4 ptrace (GETREGS, Pid,Tgt )

5 ptrace (POKETEXT, Pid,Tgt ) 6 ptrace (POKETEXT, Pid,Tgt ) 7 ptrace (DETACH, Pid,Tgt )
```

- `Pid = 57`  `Euid = 500`  `Tgt = 58`
- `Pid = 100`  `Euid = 500`  `Tgt = 101`
- `Pid = 100`  `Euid = 500`  `Tgt = 101`
Outline

1. Some Security Issues
2. Example: The `ptrace` Attack
3. Detecting The Attack, Using ORCHIDS
   - Demo
   - Under the Hood
4. The Architecture of ORCHIDS
5. Beyond: Additional Features, Further Attacks
6. Way Beyond
7. Conclusion
8. Other Things That Cannot Fit In The Talk
Efficient implementation, through compilation to bytecode trees.
A Modular Architecture

Event Input

Data Dissection Modules
- generic
- PAM
- sshd
- sendmail
- ...</p>

Raw Input Modules
- snare
- apache
- syslog
- ...</p>

Main Event Dispatcher
- textfile
- network
- sunbsm
- database
- ...

External Event Sources
- Polled I/O
- Real Time I/O
- (Files, Network, Database, Programs, ...)
- log file
- snmp
- syslog/snmp-trap
- sql request
- sql trigger

ORCHIDS
Jean Goubault-Larrecq, Julien Olivain
Issues
Example Attack
Running
ORCHIDS
Demo
Under the Hood
Architecture
Beyond
Way Beyond
Conclusion
Misc
Input Sources

Multi-sensor:

- **System** sensors:
  - SNARE, *syslog* (Unix);
  - MS EVT (Windows);

- **Network** and equipment sensors:
  - Cisco logs;
  - SNMP sensors;
  - Linux NetFilter.
  - ...

- **Meta-sensors**: e.g., SNORT used as a sensor.
- **Filters**: e.g., NetEntropy entropy tester.
- ...
Input Sources

**Multi-sensor:**

- **System sensors:**
  - SNARE, syslog (Unix);
  - MS EVT (Windows);
- **Network and equipment sensors:**
  - Cisco logs;
  - SNMP sensors;
  - Linux NetFilter.
  - . . .
- **Meta-sensors:** e.g., SNORT used as a sensor.
- **Filters:** e.g., NetEntropy entropy tester.
  - . . .

**Multi-port:** reads from UDP connections, SNMP connections, log files.
Outline

1. Some Security Issues
2. Example: The `ptrace` Attack
3. Detecting The Attack, Using ORCHIDS
   - Demo
   - Under the Hood
4. The Architecture of ORCHIDS
5. Beyond: Additional Features, Further Attacks
6. Way Beyond
7. Conclusion
8. Other Things That Cannot Fit In The Talk
Cuts (à la Prolog): generalize shortest runs. Allow one to specify removing threads thought to be irrelevant.
Cuts (à la Prolog): generalize shortest runs. Allow one to specify removing threads thought to be irrelevant.

**Green cuts:** Preserve the *no-masking* property:

Each attack (family of runs) \(\Rightarrow\) at least one alert is reported.

Shortest runs are an example, with an additional *optimality* property: exactly one alert is reported.
Cuts (à la Prolog): generalize shortest runs. Allow one to specify removing threads thought to be irrelevant.

**Green cuts:** Preserve the *no-masking* property:

*Each attack (family of runs) ⇒ at least one alert is reported.*

Shortest runs are an example, with an additional *optimality* property: exactly one alert is reported.

**Red cuts:** Forget attacks.

- Avoid congestion, avert attacks on the IDS.
- Simulate monitors. (case where no thread is spawned dynamically.)
- **Intermediate** reports. (emit them, kill threads having emitted none, then proceed.)
- Implement the *without* operator (check that no event satisfying $F$ occurs while waiting for $G$). (e.g., kill threads when monitored process exits.)
Jean, I’m afraid you did not insist on the fact that you could catch whole families of attacks by just one ORCHIDS signature rule. Have you demonstrated the \texttt{do brk} attack, by the way?
Detect Families of Attacks

Signatures can be made to match not just one attack, rather whole families.

The Entropy Checker

$q_1 \xrightarrow{\text{entropy-low} (X)} q_0 \xrightarrow{\text{ssl-error} (X)}$ detects all buffer overflow attacks on crypto protocols (ssh1, ssh2, https, ldaps, ...).

- By the way, this is a network attack. ORCHIDS is not limited to system attacks.
- By the way, it uses the Net-Entropy sensor to detect entropy anomalies in input flow.
Detect Families of Attacks

Signatures can be made to match not just one attack, rather whole families.

The Pid Tracker

detects all attacks in the style of

do_brk, mmap, munmap, mremap, etc. [MortonStarzetz’03].

- Principle: detect that some user gained root privileges without using the authorized set*id mechanism, whichever way he actually managed to do so.
- Technically, a form of dynamically-spawned monitors.
One of the most serious attacks ever actually used.

- Crackers used it to bog down the Savannah (GNU) and Debian servers in 2004. Downtime: several weeks.
One of the most serious attacks ever actually used.

- Crackers used it to bog down the Savannah (GNU) and Debian servers in 2004. Downtime: several weeks.
- Vicious attacks: give rise to no event at all.
  
  ...except a flurry of calls to `do_brk` (but this is what `malloc` calls!)
  
  ...except a flurry of `SIGSEGV` signals (not logged by SNARE!)

- Principle: rewrite the information the kernel keeps on our (user) process by mapping the kernel into the address space of the process (!).
Outline

1. Some Security Issues
2. Example: The \texttt{ptrace} Attack
3. Detecting The Attack, Using ORCHIDS
   - Demo
   - Under the Hood
4. The Architecture of ORCHIDS
5. Beyond: Additional Features, Further Attacks
6. Way Beyond
7. Conclusion
8. Other Things That Cannot Fit In The Talk
Use an embedded Prolog interpreter. Applications include:

- Keep set of attacks that have already succeeded.
- Maintain Black Lists.
- Realize that user $A$ has succeeded in the past in mounting an attack on machine $M$ giving him a user account, allowing him to try and launch a second attack on $M$ in the hope of gaining root privileges.
Information Correlation

Use an embedded Prolog interpreter. Applications include:

Reason about network topology.

- realize that 127.0.0.1 and localhost are the same machine (as in the M2D2 model [MMDD03]);
- realize that two machines $A$ and $B$ are neighbors and may have cooperated in mounting an attack.
Outline

1. Some Security Issues
2. Example: The `ptrace` Attack
3. Detecting The Attack, Using ORCHIDS
   - Demo
   - Under the Hood
4. The Architecture of ORCHIDS
5. Beyond: Additional Features, Further Attacks
6. Way Beyond
7. Conclusion
8. Other Things That Cannot Fit In The Talk
Conclusion

ORCHIDS, an efficient on-line, real-time intrusion prevention system.

- Handles real, recent, sophisticated attacks;
- Produces detailed reports, runs countermeasures; i.e., emergency measures until the sys. admin. reacts.
- Multi-sensor, multi-port, multi-event. . . multi-whatever.
Conclusion

ORCHIDS, an efficient on-line, real-time intrusion prevention system.

- Handles real, recent, sophisticated attacks;
- Produces detailed reports, runs countermeasures;
  i.e., emergency measures until the sys. admin. reacts.

- Multi-sensor, multi-port, multi-event... multi-whatever.

Care to join?  Contact:

Julien Olivain  Jean Goubault-Larrecq
olivain@lsv.ens-cachan.fr  goubault@lsv.ens-cachan.fr
Outline

1. Some Security Issues
2. Example: The `ptrace` Attack
3. Detecting The Attack, Using ORCHIDS
   - Demo
   - Under the Hood
4. The Architecture of ORCHIDS
5. Beyond: Additional Features, Further Attacks
6. Way Beyond
7. Conclusion
8. Other Things That Cannot Fit In The Talk
Exploits a buffer overflow in the OpenSSL code for Apache, implementing SSL v.2.
Effect: remote exploit, obtaining a remote shell.
The mod_ssl Attack [McDonald03]

Exploits a buffer overflow in the OpenSSL code for Apache, implementing SSL v.2.
Effect: remote exploit, obtaining a remote shell.

- Extremely complex attack.
- Exploits several Apache threads to get vulnerability information.
- Transmits vulnerability information through encrypted SSL channel.
The SSL v2 Handshake Protocol

ClientHello:
- client-cipher-list, old-id, Nc

ServerHello:
- cipher-list, cid, {cert} \(K_s\)

ClientMasterKey:
- \(\{K_m\} \ K_s, \ key_{arg}, \ldots\)

ServerVerify:
- \(\{N_c\} \ K_m\)

ClientFinished:
- \(\{cid, \ldots\} \ K_m\)

ServerFinished:
- \(\{id, \ldots\} \ K_m\)
An (Important) Detail of Implementation in OpenSSL

In OpenSSL (old version), this is an array whose bounds are *not* checked...
When the server receives `ClientMasterKey`, it copies it into:

```c
typedef struct ssl_session_st
{
    int ssl_version;
    unsigned int key_arg_length;
    unsigned char key_arg[SSL_MAX_KEY_ARG_LENGTH];
    int master_key_length;
    unsigned char master_key[SSL_MAX_MASTER_KEY_LENGTH];
    ...
} struct ssl_session_st *prev, *next;
```
struct ssl_session_st
{
    int ssl_version;
    unsigned int key_arg_length;
    unsigned char key_arg[];
    int master_key_length;
    unsigned char master_key[];
    [...] struct ssl_session_st *prev, *next;
};

if this block is free

End of struct

Size of previous block
Size of current block
Pointer to next block
Pointer to previous block
struct ssl_session_st
{
    int ssl_version;
    unsigned int key_arg_length;
    unsigned char key_arg[];
    int master_key_length;
    unsigned char master_key[];
    ...
    struct ssl_session_st *prev, *next;
};

We make the libc believe that this block is free!
struct ssl_session_st
{
    int ssl_version;
    unsigned int key_arg_length;
    unsigned char key_arg[];
    int master_key_length;
    unsigned char master_key[];
    [...]  
    struct ssl_session_st *prev, *next;
};

End of struct

Call free()

*(fwd+12)=bk

towards the GOT

towards shellcode
The rest of the attack, in short

- To retrieve the address of the shellcode, have the server retransmit all needed information by writing into the `session_id` field: we get the information, encrypted, in the `ServerFinished` message.
- We now know at which addresses the server works.
- Now replay a similar attack in a second SSL session to really execute the shellcode.
- The server now serves a connection to an `apache` or `nobody` shell through HTTP. (Now play a user-to-root attack... )
Describing the attack

Note: use an entropy input module (to be published);

Detects any attack on SSL where plain texts are served instead where we expect ciphertexts.