**Operational Programme**

“Competitiveness”

R&D Cooperations with Organizations of non-European Countries

---

**EAR: Early warning system for automatic detection of Internet-based cyberattacks**

*(Code G.S.R.T.: ΗΠΑ-022)*

---

**D3.1 “System Implementation”**

**Abstract:** This document describes the implementation of the EAR Early Warning System for Internet-Based Cyber-Attacks. The deliverable is accompanied by a Compact Disc containing the relevant source code.

---

<table>
<thead>
<tr>
<th>Contractual Date of Delivery</th>
<th>12 May 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Date of Delivery</td>
<td>12 May 2005</td>
</tr>
<tr>
<td>Deliverable Security Class</td>
<td>Public</td>
</tr>
<tr>
<td>Editor</td>
<td>Periklis Akritidis</td>
</tr>
</tbody>
</table>

The EAR Consortium consists of:

- **FORTH**
  - Coordinator
  - Greece
- **GA Tech**
  - Principal Contractor
  - USA
- **FORTHnet**
  - Principal Contractor
  - Greece
Contents

1 Introduction ........................................... 3

2 Implementation ........................................ 4
   2.1 Overview ........................................ 4
   2.2 Monitor ......................................... 4
       2.2.1 Source code layout ......................... 6
       2.2.2 Compilation-time parameters ............... 6
       2.2.3 Substring cache ............................ 6
       2.2.4 Cache entry ................................ 7
       2.2.5 Hashing of cache entries ................. 8
       2.2.6 EAR ......................................... 8
       2.2.7 Main ......................................... 9
       2.2.8 Rabin Fingerprints ......................... 9
       2.2.9 Hex printing ................................ 10
       2.2.10 Flow reconstruction ....................... 10
       2.2.11 Excluding traffic ......................... 10
       2.2.12 Reporting .................................. 10
       2.2.13 Interface with higher-level modules .... 10
       2.2.14 Act on Alerts .............................. 11
       2.2.15 Remote Monitor ............................ 11
       2.2.16 Monitor Command-Line Syntax ............ 11
   2.3 Logic ............................................. 12
   2.4 Graphical User Interface ......................... 12

References ............................................. 13
# List of Figures

2.1 System components .............................................. 4  
2.2 Monitor components .............................................. 5  
2.3 Sample alert message ............................................. 11  
2.4 Sample status message .......................................... 11  
2.5 Monitor command-line syntax .................................... 12  
2.6 Contagion: The worm appears in both incoming and outgoing traffic of a host. ............................................. 12  
2.7 Issued alerts and inspection of the detected worm substring. ............................................. 14  
2.8 Issued alerts and inspection of the flows that carried the offending substring. ............................................. 14  
2.9 The EAR statistics panel. .......................................... 15  
2.10 The EAR configuration dialog. ..................................... 15
Chapter 1

Introduction

In this document we discuss the implementation of the EAR Early Warning System for Internet-Based Cyber-Attacks. The deliverable is accompanied by a Compact Disc containing the relevant source code.

The system operates by passively monitoring network traffic and identifying strings that belong to worms and can be used as signatures for filtering worms. The main objectives of the system are (a) to detect previously unknown Internet worms, (b) to detect them in time, (c) to detect them without false positives, and (d) to detect them without human intervention.

The system operates by detecting substrings in network stream contents that are sent to more than a number of destination hosts within a certain period of time. A fundamental condition for the system to detect a worm is that the attack must contain an invariant string that can be used as a signature. A number of additional heuristics employed to boost performance and reduce false positives are described.

During the course of this project several research articles were published. Efficient content-based fingerprinting of zero-day worms [1], published in the IEEE International Conference on Communications (ICC 2005), describes the repetitive-content-based worm detection heuristics. STRIDE: Polymorphic Sled Detection using Instruction Sequence Analysis [2], published in the 20th IFIP International Security Conference (IFIP/SEC 2005), discusses the detection of polymorphic attacks. Design and Implementation of a High-Performance Network Intrusion Prevention System [4] and Generating Realistic Workloads for Network Intrusion Detection Systems [3], study mechanisms for enforcing the signatures generated by the worm detection system.
Chapter 2

Implementation

In this chapter we discuss the implementation of the EAR Early Warning System for Internet-Based Cyber-Attacks.

2.1 Overview

EAR was implemented on the GNU/Linux platform, but it is designed to be portable to other platforms, including Microsoft’s.

EAR is composed of three major modules: a high-performance monitor, programmed in the low-level C language for performance, implements the main filter; generated alerts are processed by a second component, programmed in the high-level Python language, for flexibility; finally, the Graphical User Interface (GUI) is implemented in Python and GTK. Figure 2.1 shows a component diagram.

In the following sections we describe each of the three components.

2.2 Monitor

The Monitor module is implemented in C, for performance reasons. It communicates with the Logic module by emitting records in ASCII format on its standard output.
It has several submodules, which are illustrated in Figure 2.2, and are described below and in the following subsections.

**main** This module drives the rest of the system. It uses the **Options** module to parse the command-line arguments and the Libnids library to capture network traffic. It drives the **EAR** module, which is the API of the worm detection system.

**options** This module is responsible for parsing command line arguments, and is used by the **Main** module.

**libnids** This is a third-party library for TCP flow reconstruction.

**ear** This module implements and provides an API to the EAR worm detection algorithms. It uses the **Cache**, **Rabin**, and **Report** modules.

**cache** This module is responsible for the storage of substring fingerprints, and implements the appropriate data-structures.

**rabin** This module is responsible for computing substring fingerprints using the Rabin algorithm.

**report** This module is responsible for reporting results on the standard-output.

![Monitor components diagram](image)

Figure 2.2: Monitor components
2.2.1 Source code layout

Here we describe the files that make up the source code of the Monitor.

cache.h and cache.c These files implement the substring cache (Cache module).
ear.h and ear.c These files contain the ear API (EAR module).
report.h and report.c These files handle reporting (Report module).
main.c Source file containing the main function and libnids calls (Main module).
options.h and options.c Handling of command-line arguments (Options module).
rabin.h and rabin.c Implementation of incremental Rabin fingerprints (Rabin module).

The following files contain helper routines possibly used by many modules.

cfg.h This file contains parameters tunable during compilation-time.
timer.h Inline timer manager functions (xtimeradd, xtimersub, xtimercmp)
cputime.h and cputime.c CPU time utilities.
hex.h and hex.c These files contain hex printing utility functions

2.2.2 Compilation-time parameters

The compile time parameters are configured in config.h which is included from other source files. They include:

MAX_DST This parameter controls the size of the table allocated for storing multiple destinations. The destinations threshold used for triggering an alert is tunable at runtime, but it cannot exceed this value.

VERBOSE This flag controls whether additional information is stored, such as the source addresses, ports, and offsets of the captured substrings. If this flag is not set, then the corresponding fields in the reported results are set to "N/A".

2.2.3 Substring cache

The substring cache is responsible for storing information about the fingerprints of encountered substrings. It is implemented in files cache.h and cache.c.

The substring cache is represented by the struct cache ADT, which is a container of cache entries, represented by struct cache_entry. struct cache supports the following operations:
struct cache *cache_create(int capacity, int threshold)  Creates a cache that will store information about substrings for a number of msec given by the capacity argument. The threshold parameter corresponds to the distinct destinations threshold.

void cache_destroy(struct cache *cache)  Frees the resources used by the cache’s data structures.

int cache_usage(struct cache *cache)  Returns the number of substrings currently tracked by the cache.

struct cache_entry *cache_entry_create(struct cache *cache, u_int32_t hash)  Create and returns a cache entry initialized with the given hash, clean flags, destination count of zero and set expiration time.

struct cache_entry *cache_entry_lookup(struct cache *cache, u_int32_t hash)  Looks up a cache entry by its substring fingerprint.

void cache_touch(struct cache *cache, struct cache_entry *cache_entry)  Renews the expiration time of the given cache entry.

The current time is maintained using the timestamp of the last processed packet. This way the system works reliably with traces and a syscall is avoided. The CURRENT_TIME macro provides this information.

2.2.4 Cache entry

Each cache entry contains the 32-bit fingerprint of the corresponding substring, the time when the entry will expire, the distinct destination count, the destinations, and some flags. If the VERBOSE compile-time flag has been defined, the cache entry also contains the destination ports, the sources, the source ports, and the flow offsets of the substrings. The entry also contains a synonym_count field, used for hashing.

A single destination can be stored within the entry, but for more than one destinations, an additional table is allocated and used (struct cache_dest). So the cache entry also contains a pointer to the extra table, but this pointer could be allocated as a C union with the single destination stored within the entry, as the two variables are never used at the same time.

Each cache entry can have one or more of the following flags set:

CACHE_EMPTY  The slot is unused.

CACHE_MULTIPLE_DST  Substring has multiple destinations stored in extra table.

CACHE.Track  Substring should be tracked.

CACHE_IGNORE  Substring should be ignored.
CACHE_PIN  Substring should not be evicted

2.2.5 Hashing of cache entries

The cache entries are stored in a hashtable in the struct cache data structure.

The hashtable uses linear probing as a collision resolution scheme. To determine whether a slot can be reused, the current time is compared against an entry’s time of insertion. Therefore, old entries are implicitly evicted after a time out.

More specifically, a slot can be used if it is empty (the CACHE_EMPTY flag is set) or it has been expired. A entry is considered expired if it has been for long enough in the cache and the CACHE_PIN flag is not set.

The following hash statistics are maintained by cache_entry_lookup(), and printed in status reports:

hash_lookups  The number of hashtable lookup operations.

hash_probes  The number of probes (a lookup may require multiple probes in the event of collision).

hash_collisions  The number of hash functions collisions.

2.2.6 EAR

The EAR module is the interface to the worm detection algorithms. It provides the following operations:

struct ear *ear_create(int span, int capacity, int dst_threshold, int stream_limit, uint32_t sampling_mask)

Creates an instance of the monitor.

void ear_process(struct ear *ear, struct tcp_stream *a_tcp, char *data, int dsize, int offset, struct ear_flow *)

Process a chunk of data.

void ear_destroy(struct ear *)

Destroys an instance of the monitor.

struct ear_flow *ear_flow_create(void)

Creates a flow state object. The ear_process() function requires such an object for storing per flow state.

void ear_flow_destroy(struct ear_flow *)

Destroys a flow state object.

The ear_create() function takes the following parameters:

span  The substring size.

capacity  The capacity of the cache in milliseconds.

dst_threshold  The distinct destination threshold.

stream_limit  The size of the portion of the flows that is processed.
The substring sampling mask. The system will ignore substrings whose fingerprints do not match the sampling mask.

Whether to process substrings that contain ASCII nul characters.

The **ear_process()** function accepts a data buffer and a stream descriptor (**struct tcp_stream**, defined by libnids). It maintains statistics (bytes processed, samples memory usage) and computes incremental rabin fingerprints.

Individual fingerprints are further processed by the **process_hash()** function, which performs deterministic sampling, and then looks up the substring in the cache. If the hash is found, and it is not ignored (**CACHE_IGNORE** flag is not set), and the current destination has not been recorded in the cache entry, then it records the current destination and renews the expiration time of the entry. It also allocates and populates a table for storing multiple destinations if that has not been done already.

If the distinct destinations threshold has been reached, the **trigger()** function is called, which involves the **Report** module.

If the hash is not found in the cache, a cache entry is created and the substring destination is recorded.

If the **VERBOSE** compile-time option is set, additional information is recorded together with the destination.

### 2.2.7 Main

The main function creates a struct ear instance using the command line arguments parsed by the options module, and then registers a tcp callback with libnids and invokes libnids. The callback instructs libnids to collect only traffic sent to TCP servers. It invokes **ear_process()** for each data portion captures and instructs libnids to buffer substring length − 1 bytes.

The options module provides a function to parse command line arguments, and exports variables with option values.

### 2.2.8 Rabin Fingerprints

To compute rabin fingerprints, we use a state common for all flows and a separate hash for each flow. The common state includes a lookup table and the shift out multiplier. The hash has to be initialized to zero. The following functions are provided:

- **rabin_t *rabin_create(int span_size)** Creates the common state.

- **SHIFT_IN(rabin, hash, c)** Adds a byte to a hash.

- **SHIFT_OUT(rabin, hash, c)** Removes a byte from the hash.
2.2.9 Hex printing

The following utility functions are used to print binary data:

```c
void hex_print(unsigned char *buf, int buflen, char *label)  // Print a chunk of data in hex to standard output and prepend it with a label.
void hex_print_f(FILE *f, unsigned char *buf, int buflen, char *label)  // Print a chunk of data in hex to given file and prepend it with a label.
int hex_parse(char *hex, unsigned char **bufp)  // Parse hex representation, return the length of the resulting data, and set pointer argument to a buffer with the data, allocated with malloc().
```

2.2.10 Flow reconstruction

For flow reconstruction we use the libnids library which emulates the IP stack of Linux 2.0.x. Libnids offers IP defragmentation, TCP stream assembly and TCP port scan detection. The library is available from [http://libnids.sourceforge.net/](http://libnids.sourceforge.net/).

2.2.11 Excluding traffic

Libnids is instructed to process only traffic sent from clients to servers. In addition, from each flow only the first X bytes are processed. X is configurable, and typical values include 1K, 10K, 100K.

2.2.12 Reporting

The Report module provides the following functions used for reporting alerts, attacks, and statistics:

```c
void report_alert(struct ear *ear, struct ear_result *result, struct cache_entry *cache_entry)
void report_attack(struct ear *ear, struct tcp_stream *a/tcp)
void report_stats(struct ear *ear)
```

2.2.13 Interface with higher-level modules

The interface between the monitor and the high-level components is ASCII-based. Messages are headed by a keyword followed by message-dependend contents. An alert message is denoted by the ALERT keyword, followed by a line with space delimited data items including the hash of the string that triggered the alert. Figure 2.5 shows a sample alert message.
2.2.14 Act on Alerts

The functionality of acting on an alert is implemented by passing the alert message to a shell script. The script can process and forward the alert to external mechanisms, such as content filtering firewalls.

2.2.15 Remote Monitor

It is often the case that the Monitor must run on a remote server, but the GUI must run on an operator’s computer. This functionality is supporting by using the `rear.sh` script instead of the actual monitor executable, which uses SSH to control a remote ear monitor instance.

2.2.16 Monitor Command-Line Syntax

The Monitor module produces a single executable called `ear`, that has the following command-line syntax:
Usage: ear OPTIONS [ filter expression ]
   -f, --offset=INT     flow offset threshold
   -p, --period=INT     period threshold
   -s, --select-mask=HEX select mask
   -t, --targets=INT    targets threshold
   -l, --length=INT     substring length
   --skip-nul           skip strings with ASCII nul characters
   -n, --home-net=NET   network under protection
   -r file              read packets from file
   -i interface         capture packets from interface
   -h, --help           display this help message

Figure 2.5: Monitor command-line syntax.

2.3 Logic

The alerts that are issued by the high-performance monitor, are further processed by the Logic module. The following checks are currently supported:

Sources threshold  The number of distinct sources appearing to spread the attack is required to be above a threshold.

Contagion threshold Contagion is the number of targets that also appear to act as sources, further propagating the attack. It is required that this number is above a threshold. Figure 2.6 illustrates this.

Figure 2.6: Contagion: The worm appears in both incoming and outgoing traffic of a host.

The polymorphic sled detection algorithm (STRIDE) that has been designed in the context of the EAR project, will be eventually integrated here as well.

2.4 Graphical User Interface

We have provided a Graphical User Interface (GUI) for the management of the system by operators. The GUI allows the configuration of the monitor’s runtime parameters, and provides a listing of issued alerts.

The main screen of the GUI shows the alerts issued so far. By selecting an alert it is possible to inspect the offending substring, as shown in Figure 2.7. It is also possible to inspect the flows that carried the substring and the offset of the substring within each flow, as shown in Figure 2.8.
The statistics maintained by the monitor can also be inspected, as shown in Figure 2.9. The following statistics are provided:

**Last updated**  The time that the statistics were last updated.

**Current usage**  The number of currently tracked substrings.

**Maximum usage**  The maximum memory usage.

**Average usage**  The average memory usage.

**Utilization**  The CPU time/ Wallclock time ratio over the last interval.

The monitor parameters can be configured by selecting the EAR Configuration menu entry from the Tools menu. It is possible to configure the following parameters, as shown in Figure 2.10:

**Min. Targets**  A worm must have target at least this many targets.

**Min. Length**  A worm must have at least this length.

**Max. Offset**  A worm must occur at most at this offset within its flow.

**Max. Period**  A worm must occur with a period at most equal to this value.

**Filter**  A filter that is applied on network traffic before it is processed. It can be usefull to exclude certain hosts.

**Tracking Mask**  This value control the deterministic sampling rate.

**Skip substrings with ASCII nul**  Whether to process substring with the ASCII nul character.

**Min. Sources**  A worm must be spread by at least this many sources.

**Min. Contagion**  The value of the contagion threshold.

**Home Net**  The network under protection.
Figure 2.7: Issued alerts and inspection of the detected worm substring.

Figure 2.8: Issued alerts and inspection of the flows that carried the offending substring.
Figure 2.9: The EAR statistics panel.

Figure 2.10: The EAR configuration dialog.
References


