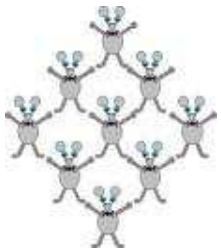

M-ICE
Modular Intrusion Detection and Countermeasure Environment

Developer Guide

Version 0.1

Changelog

Version	Date	Author	Changes, Problems, Notes
0.1	2003−12−31	Thomas	− initial version

Copyright and Licence Notes

Abstract

On the following pages you will find everything you need to develop your own M-ICE modules and to improve the existing code. The example systems are running SUSE LINUX 9.0 Professional on i386 architecture. [1]

[1] I am not responsible for any damage caused by my software or by this documentation!

Contents

Chapter 1

Architecture

Before we go to the details of the systems architecture I should make clear the goals I tried to reach. First I want to provide a system that can be customized easily to fit everybodys needs and to close the gap between IDS research and IDS application in real-life.

M-ICE is splited up into several basic components. These basic components can be customized by applying dynamic loadable modules. All components exchange information via a TCP/IP network or via local interprocess communication. All parts of M-ICE can be run on a single system or can be installed on a dedicated system and distributed in a network.

First our IDS needs to get the audit-data. These data is generated on a host, the client in the IDS context, and can be application-level data like syslog or kenel-level data like syscall logs (BSM, LAuS, ...). Forwarding these data will by done by the `DataForwarder`. The `DataForwarder` has to load three modules, one to filter the raw data, one to pseudonymize the user related identifiers, and another one to convert the audit-data to a standard format. After passing all modules the `DataForwarder` sends the data to an analysis-agent and/or to a database.

M-ICE uses two kinds of databases, one for archiving raw audit-data and another one for collecting all detected alerts. In the case of uncirtainness about on system compromise the first databases can be queried manually to obtain useful informations.

The raw log-database is assembled by using a gerneric network daemon that is called `BufferDaemon` and two modules. One module is used to decode the data received from the network, the other module processes the network data after a user-defined delay and send it to a local MySQL database.

The analysis-agent is assembled in an equal way by using the `BufferDaemon`, a decoding module, and a postprocessing module. The postprocessing module is the code part where the analysis happens. The analysis-agent can send its data to another analysis-agent and/or to the `ManagementHost`.

The `ManagementHost` is a combination of a `BufferDaemon`, a decoding mod-

ule, a postprocessing module, several reaction-agents, and a MySQL database for archiving received alerts. The `ManagementHost` receives the alerts of the M-ICE analysis-agents in the IDMEF [1] format. By using a standardized message format in combination of a flexible relation of alert-IDs and reaction-types it is possible to handle even other Intrusion Detection Systems like Snort (with the apropriate output module).

If a reaction has to be executed, a message (RPC like) will be send from the `ManagementHost` to the `ReactionDaemon` on the client. The reactions are implemented as a dynamic loadable module. The modules are identified by an ID that is send within the message from the `ManagementHost`. By this way reactions can be added at runtime and are very flexible.

For more information you should read the Admin Guide.

Chapter 2

Source Code

The source-code of M-ICE was developed over one and a half year [1] and contains more then 75.000 lines of code. I tried to keep the code clean and secure. But to be honest it often becomes too complex and the cryptographic part is not very secure, because to my lack of knowledge about cryptography at the time of development. Hopefully I did not scare you away at this point.

Due to the fact that this promissing project becomes too huge to be handled by one person and to share knwoledge and fun I wrote this paper to provide all interested developers with an overview of the M-ICE internals.

2.1 CVS

The source-code is managed by Sourceforge's CVS server. CVS is a good choice for working on files with different persons at the same time but doesn't provide any serious security measurements needed in non-trusted environments.

Nevertheless it's easy to use and widely available and I do not want to scare away you, the developer. Trust comes before verification!

For developers not familiar with CVS I'll provide a (very) short introduction. First you need to set some environment variables (bash-like) before you can check out the source-code.

```
> export CVSROOT="<yourlogin>@cvs.sourceforge.net:/cvsroot/m-ice"
> export CVS_RSH=ssh
> cvs checkout m-ice
```

After you changed and tested a file you can submit it to the CVS repository and comment your changes.

```
> cvs commit main.c
```

For directory sub-trees.

[1] Sometimes under heavy influence of coffeine and beer to survive long nights

```
> cvs commit -R
```

Sometimes it happens that another developer edits the same file as you so there is a conflict with the file. Therefore you have to fetch the new version of the file and merge you modifications.

```
> cvs update main.c
```

To update full directory structures use:

```
> cd m-ice
> cvs update -PRd
```

To add new files or directories use the following command.

```
> cvs add mymain.c
> cvs commit mymain.c
```

These are really basic commands, but it is enough to handle your daily work. Stress your favorite internet search engine to find some web sites describing CVS in more detail like the “CVS Book“ [10].

2.2 Source Tree

After checking out the source-code of M-ICE you will see a (somehow) logical ordered file and directory structure.

```
> ls -l m-ice/
insgesamt 58
-rw-r--r--    1 thetom   users           0 2003-09-09 21:23 AUTHORS
drwxr-xr-x    2 thetom   users         128 2003-09-18 12:32 CVS
-rw-r--r--    1 thetom   users       36880 2003-09-18 12:32 ChangeLog
-rw-r--r--    1 thetom   users          77 2003-09-09 21:23 INSTALL
-rw-r--r--    1 thetom   users         360 2003-09-09 21:23 Makefile.am
-rw-r--r--    1 thetom   users           0 2003-09-09 21:23 NEWS
-rw-r--r--    1 thetom   users           0 2003-09-09 21:23 README
-rw-r--r--    1 thetom   users         132 2003-09-09 21:51 TODO
drwxr-xr-x    4 thetom   users         136 2003-09-09 21:33 client
-rw-r--r--    1 thetom   users        1659 2003-09-09 21:23 configure.ac
drwxr-xr-x    3 thetom   users        1312 2003-09-09 21:34 etc
drwxr-xr-x    4 thetom   users         304 2003-09-09 21:34 include
drwxr-xr-x    3 thetom   users         584 2003-09-09 21:34 init-scripts
drwxr-xr-x    3 thetom   users         408 2003-09-09 21:34 libs
drwxr-xr-x    9 thetom   users         256 2003-09-09 21:37 modules
drwxr-xr-x    3 thetom   users         184 2003-09-09 21:38 mysql
drwxr-xr-x    7 thetom   users         264 2003-09-09 21:38 reaction-agents
drwxr-xr-x    5 thetom   users         168 2003-09-09 21:39 server
```

Let's call the checked out m-ice directory the root directory in our context. The root directory contains the basic files needed for the `autoconf` [11] tools and the the directories that hold the corresponding source-code files.

client	contains the `DataForwarder` that runs on the monitored host
etc	all configuration files are located here
include	various include files
init-scripts	sysV init-scripts for the different programs
libs	libraries
modules	all modules for the `DataForwarder`, the `ReactionDaemon` and `BufferDaemon`
mysql	just two scripts to create the MySQL databases
reaction-agents	includes all reaction-agents for the `ManagementHost`
server	`BufferDaemon` and `ReactionDaemon`

2.3 Autoconf

The GNU autoconf tools are well known and widely used in the open-source software community. They are very helpful when developing on different architectures or when building binary packages like RPM files or DEB files.

First you have to install the common autoconf files, by running `autoreconf`.

```
> autoreconf --install
configure.ac: installing `./install-sh'
configure.ac: installing `./mkinstalldirs'
configure.ac: installing `./missing'
Makefile.am: installing `./COPYING'
client/dataforwarder/Makefile.am: installing `./depcomp'
```

For checking and setting the environment the `autoreconf` tool creates the `configure` script by reading the `configure.ac` file. Run `configure --help` to get a list of commands that are recognized, but mostly the default setting will suffice. For M-ICE call:

```
> ./configure --prefix=/usr --sysconfdir=/etc
[check environment and set variables]
```

After the script finishes successfully `include/config.h` is created and includes all environment-specific variables. Additionally all `Makefiles` in all sub-directories are created as well.

All this is very simple, believe me.[2] Just read the `configure.ac` and `Makefile.am` files and search the Internet.

[2] because even I was able to understand it ;)

2.4 Compiling and Installing

To compile the source just type `make`, `make install` installs the files in the system. To clean everything up just execute `make clean`.

That was short, thanks to GNU. :)

Chapter 3

Puzzle Parts

All this may look a bit complex at the first time because of the various programs loading different modules. The modules M-ICE uses are nothing more then shared libraries opened by a process with a `dlopen()`-wrapper function. Modules are used very often in the M-ICE framework just to keep flexibility and extensibility high.

In this chapter I try to put the puzzle parts together to show you the big picture.

3.1 Data Forwarder

The `DataForwarder` watches and reads raw log-data from the client system, filters the data, strippes off user-identifiers and formats the data. These tasks are handled by modules. Additionally the code is split up to let only a few code parts run with root privileges. In the future these privileges should even be more restrictive by using Linux' Capabilities.

3.1.1 Architecture

The `DataForwarder` splits up in three processes. The main process handles the configuration and the communication between the other M-ICE components. The log-watching process keeps track of log-files that the filedescriptor-server (only process running as root) opened.

The log-watching process asks the filedescriptor-server to open a file and receives the appropriate filedescriptor. The whole communication is done via a unix-domain-socket.

After receiving the filedescriptor the process starts looking for changes of each file, reads new data line-by-line, puts it in a shared-memory segment and handles a semaphore.

The main process waits for new data being put in the shared-memory segment. The new log-data is processed by the three modules before it is send away for further analysis and storing.

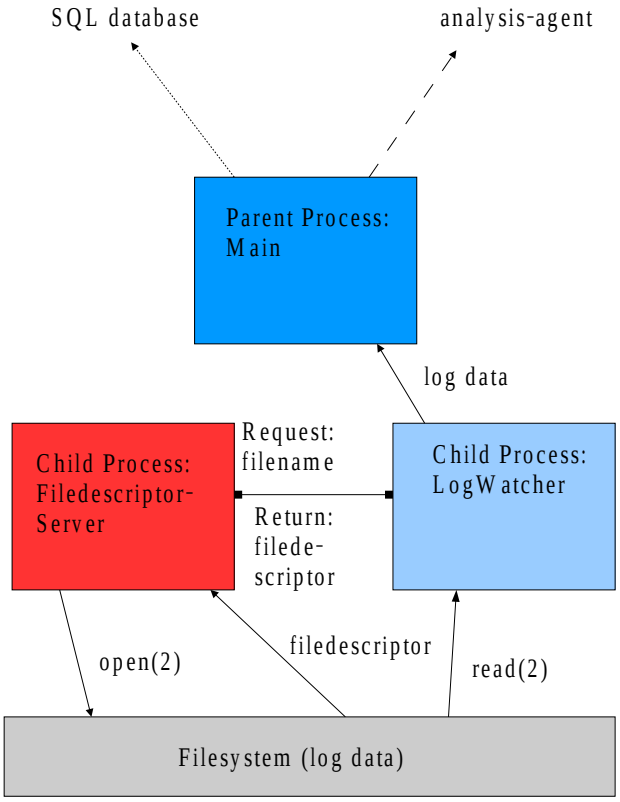

Figure 3.1: `DataForwarder` Architecture

3.1.2 Handling the Modules

The `DataForwarder` opens every module and looks up the corresponding symbols. Symbols are function names. Pointers to these functions are used by the `DataForwarder` to call these functions. The following code snippet shows it (debug code is remove to improve readability).

```
/* INIT. */
lt_dlinit();

/* OPEN THE MODULE */
dlHandleFilter = lt_dlopenext(cModPathName);
```

```
/* GET THE SYMBOLS */
modFilterInit = (int(*)(char *))lt_dlsym(dlHandleFilter, SYMNAME_INIT);
modFilterFunc = (int(*)(char *, size_t))lt_dlsym(dlHandleFilter, SYMNAME_FUNC);

/* CALL INIT FUNCTIONS */
(*modFilterInit)(CfgModConfFilter[iSectModConf];
```

The `DataForwarder` loads three modules, the filter module to filter and reduce the raw log-data, the pseudonymization module to remove user identifiers for privacy reasons and a format module to convert the raw-log data to a more standardizes format.

The code you see above is used to load the filter module. First, before a `lt_`-functions can be used [1], `lt_dlinit()` has to be called.

After opening the binary module with `lt_dlopenext` the symbols (functions) of the module have to be looked up by name using `lt_dlsym`. The function is associated with a function-pointer that can be used to call the function. The `DataForwarder` does not care how the data is processed by the module. The only thing the module designer has to keep in mind is to use the standarized arguments and return values. Chapter "Structures, Protocols and Interfaces" describes this in more detail

3.2 Buffer Daemon

The `BufferDaemon` uses the same technique as the `DataForwarder` to load the modules. To handle new connections without disturbing the rest of the `BufferDaemon` the function `voidHandleClientRequest()` runs in a separat thread, reads data from a socket and hands the data over to the decoding module.

```
while( (ElemRead =
        fread(cEncodedMsg, ProcQ[ProcQEnt].Encoding.MsgSize, 1, Sock))
== 1  )
{
  /*
  ** Call Encode-Modules Decode-Function to get Plaintext Message
  */
  if( (cDecodedMsg = (*ProcQ[ProcQEnt].Encoding.FuncPtr)(cEncodedMsg,
                       ProcQ[ProcQEnt].Encoding.MsgSize)) == NULL)
  {
    log_mesg( WARN, "%s: Error while decoding received Data!\n", cProgname);
    continue;
  }
  [...]
```

[1] These are `dlopen(2)` wrapper functions [9] to make the code more portable.

The decoding module returns plaintext data in a char-array. This data is written to a ringbuffer. (Please note the handling of the mutex-lock for the data shared between different threads.)

```
  [...]
  /*
  ** Now write the P(l)aintext Message to the Ringbuffer
  */
  pthread_mutex_lock(&ProcQ[ProcQEnt].Ringbuffer.rb_mutex);

  if
  (
    intRBWrite(&ProcQ[ProcQEnt].Ringbuffer, cDecodedMsg,
                ProcQ[ProcQEnt].Output.MsgSize, TRUE) < 0
  )
    log_mesg( WARN, "%s: Error: intRBWrite()!\n", cProgname);

  pthread_mutex_unlock(&ProcQ[ProcQEnt].Ringbuffer.rb_mutex);
}  // while(fread())
```

Every `voidHandleClientRequest()` thread has a corresponding timer thread that is implemented in the function `voidTimer()`. After an user-specified time-interval these function reads the plaintext data from the ringbuffer and processes it with the help of the post-processing module [2].

```
while(TRUE)
{
  /*
  ** Sleep for Time Interval Seconds
  */
  sleep(ProcQ[ProcQEntry].Output.TimeInv);


  /*
  ** Read Data from Ringbuffer
  */
  pthread_mutex_lock(&ProcQ[ProcQEntry].Ringbuffer.rb_mutex);

  while
  (
    intRBRead(&ProcQ[ProcQEntry].Ringbuffer, cMsg,
               ProcQ[ProcQEntry].Output.MsgSize, TRUE) != -1
```

[2]The term "Output" was the old name used for the post-processing modules. This will change in one of the next versions.

```
  )
  {
    /*
    ** Call Module Function
    */
    if
    (
      (*ProcQ[ProcQEntry].Output.FuncPtr)(cMsg,
        ProcQ[ProcQEntry].Output.MsgSize) < 0
    )
      log_mesg( WARN, "%s: Debug: Error\n", cProgname);

    memset(cMsg, 0, ProcQ[ProcQEntry].Output.MsgSize);
  }

  pthread_mutex_unlock(&ProcQ[ProcQEntry].Ringbuffer.rb_mutex);
}
```

3.2.1 Decoding Module

Here is an example of an decoding module for IDMEF messages.

```
char *mice_mod_dec_idmef_twofish_LTX_func(char *cData, size_t DataSize)
{
  static char     cDecodedMsg[sizeof(CipherIdmefMsg)];
  u_short         sChkSum_Orig;
  u_short         sChkSum_New;
  register int    iCnt;
  CipherIdmefMsg  *CMsg;
  IdmefMsgFormat  *IdmefMsgPtr;


  CMsg = (CipherIdmefMsg *)  cData;


  if( CMsg->CipherTextLen != sizeof(IdmefMsgFormat) )
  {
    log_mesg(WARN, "%s: Error: Length of received Messages does not match
                    the expected Length!",
                    _mice_mod_dec_idmef_twofish_cProgname);
    return(NULL);
  }
```

```
/*
** Message is NOT decryted.
*/
if(CMsg->IVLen == 0)
{
  if(_mice_mod_dec_idmef_twofish_iDebug)
    log_mesg(WARN, "%s: Debug: Message is NOT decrypted!",
                    _mice_mod_dec_idmef_twofish_cProgname);

  IdmefMsgPtr  = (IdmefMsgFormat *)  CMsg->cCipherText;

  return(IdmefMsgPtr->cIdmefMsg);
}


/*
** Message is decryted. Let's decrypt it!
*/
if(mcrypt_generic_init(_mice_mod_dec_idmef_twofish_CryptoInfo.CryptModule,
                       _mice_mod_dec_idmef_twofish_CryptoInfo.cKey,
                       _mice_mod_dec_idmef_twofish_CryptoInfo.KeySize,
                       CMsg->IV) < 0)
{
  log_mesg(WARN, "%s: Error while initializing Crypto Module\n",
                  _mice_mod_dec_idmef_twofish_cProgname);
  return(NULL);
}

memset(cDecodedMsg, 0, sizeof(cDecodedMsg));
memcpy(cDecodedMsg, CMsg->cCipherText, CMsg->CipherTextLen);


for(iCnt = 0; iCnt < CMsg->CipherTextLen; iCnt++)
  mdecrypt_generic(_mice_mod_dec_idmef_twofish_CryptoInfo.CryptModule,
                   &cDecodedMsg[iCnt], 1);

IdmefMsgPtr = (IdmefMsgFormat *) cDecodedMsg;


/*
** Verify Checksum (CRC)
*/
```

```
  sChkSum_Orig          = IdmefMsgPtr->sChkSum;
  IdmefMsgPtr->sChkSum  = 0;
  sChkSum_New           = in_chksum((u_short *) cDecodedMsg, CMsg->CipherTextLen);

  if(sChkSum_Orig != sChkSum_New)
  {
    log_mesg(WARN, "%s: Debug: Checksum does not match... skipping Message\n",
                   _mice_mod_dec_idmef_twofish_cProgname);
    return(NULL);
  }

  IdmefMsgPtr->sChkSum = sChkSum_Orig;

  if(mcrypt_generic_deinit(_mice_mod_dec_idmef_twofish_CryptoInfo.CryptModule) < 0)
    log_mesg(WARN, "%s: Error while clearing Crypto Module!",
                   _mice_mod_dec_idmef_twofish_cProgname);

  return(IdmefMsgPtr->cIdmefMsg);
}
```

3.2.2 Postprocesssing Module

A very simple example of a post-processing module is listed below:

```
int mice_mod_syslog_LTX_init(char *ConfigFile)
{
  log_open("mice_mod_pop_syslog", LOG_PID, LOG_USER);
  return(EXIT_SUCCESS);
}

int mice_mod_syslog_LTX_func(LogFormat LogFmt)
{
  log_mesg(WARN, "Host: %s | OpSystem: %s | Release: %s | Version: %s |
                  Date: %s | Time: %s | Logline: %s\n", LogFmt.cHost,
                  LogFmt.cOSystem, LogFmt.cRelease, LogFmt.cVersion,
                  LogFmt.cDate, LogFmt.cTime, LogFmt.cLogLine);

  return(EXIT_SUCCESS);
}
```

All the module does is writing a message to the `syslogd` service.

3.3 Reaction Agents

Simple tools, called reaction-agents, are used to process the reaction-messages on the `ManagementHost`. The structure of these message is described in more detail in chapter "Structures, Protocols and Interfaces". The following text illustrates a simple example that just sends the alert-information to the `syslogd` service.

```
int main(void)
{
  FILE          *streamFifo;
  RIDMsgFormat  RIDmsg;


  log_open("rid_1_write_to_syslog", LOG_PID, LOG_NOTICE);

  if(HandleConfFile(CONFFILE) < 0)
    log_mesg(FATAL, "M-ICE Syslog: Error while parsing Config File");

  if( (streamFifo = fopen(CfgPipe.cPipe[CfgPipe.iSectionNr], "r")) == NULL)
    log_mesg(FATAL, "M-ICE Syslog: Error while opening FIFO");

  while(TRUE)
  {
    clearerr(streamFifo);
    if(fread((char *) &RIDmsg, sizeof(RIDMsgFormat), 1, streamFifo) != 1)
    {
      if(ferror(streamFifo))
        log_mesg(WARN_SYS, "M-ICE Syslog: Error while reading from FIFO | ");

      sleep(2);
      continue;
    }

    log_mesg(WARN, "M-ICE Syslog: AlertID = %s | AlertID Description = %s",
                    RIDmsg.cAlertID, RIDmsg.cAlertIDDesc);
  }

  exit(0);
}
```

Note that the reaction-message also includes the IDMEF message that we do not use here for our example. More on this later,

3.4 Reaction Daemon

The `ReactionDaemon` is run on the client too, beside the `DataForwarder`. The `ReactionDaemon` is not mandatory but is needed if you want to execute reactions on the client as a countermeasure for detected attacks. Basically the `ReactionDaemon` is a stripped down `BufferDaemon` that supports RPC-like functions and a module-type that implements the needed reaction. The reaction-module is identified by the so called function-ID.

```
/*
** Check Mode and
** Process desired Action
*/
switch( (u_int) ntohl(RctMsgPtr->Mode) )
{
  case MID_EXEC:
    iRetVal = intHandleExec(RctMsgPtr->ModeData.Exec);
    break;

  case MID_SHOW: // XXX NOT SUPPORTED
    iRetVal = intHandleShow(RctMsgPtr->ModeData.Show);
    break;

  case MID_CHECK: // XXX NOT SUPPORTED
    iRetVal = intHandleCheck(RctMsgPtr->ModeData.Check);
    break;

  default:
    iRetVal = RID_UNKNOWNMODE;
}


/*
** Send Return Value back to Client
*/
memset(SrvMsg, 0, sizeof(SrvMsg));

CipMsgPtr                 = (stCipherRctMsg *)  SrvMsg;
RctMsgPtr                 = (stReactionMsg *)   CipMsgPtr->cCipherText;
RetMsgPtr                 = (stRetvalMsg *)     &RctMsgPtr->ModeData.Retval;

CipMsgPtr->CipherTextLen  = sizeof(stReactionMsg);
CipMsgPtr->IVLen          = 0;
```

```
RetMsgPtr->ret_val    = (int)     htonl(iRetVal);
RctMsgPtr->Timestamp  = (time_t)  0;
RctMsgPtr->Mode       = (u_int)   htonl(MID_RETVAL);
RctMsgPtr->sChkSum    = 0;
RctMsgPtr->sChkSum    = in_chksum((u_short *) CipMsgPtr->cCipherText,
                        sizeof(CipMsgPtr->cCipherText));

if(writen(CliInfo.iCliSock, SrvMsg, sizeof(SrvMsg)) != sizeof(SrvMsg))
{
  log_mesg(WARN_SYS, "%s: Error: writen(SrvMsg) | Syserror\n", cProgname);
  break;
}
```

3.4.1 Reaction Module

Just pasting source-code isn't very creative but it is informative.

```
/*
** Module Functions
*/
size_t mice_mod_rct_dummy_LTX_init(char *ConfFile)
{
  //log_open("mice_mod_rct_dummy", LOG_PID, LOG_USER);

  if(_mice_mod_rct_dummy_CfgDone != FALSE)
  {
    log_mesg(WARN, "mice_mod_rct_dummy: Do NOT call init function twice, call close
    return(-1);
  }

  if(_mice_mod_rct_dummy_HandleConfFile(ConfFile) < 0)
    return(-1);

  _mice_mod_rct_dummy_CfgDone = TRUE;

  return(1);  // only one argument
}


int mice_mod_rct_dummy_LTX_func(char *cArg, size_t ArgLen)
{
  return(system(cArg));
```

```
}

int mice_mod_rct_dummy_LTX_close(void)
{
  _mice_mod_rct_dummy_CfgDone = FALSE;

  return(0);
}


/*
** Handle Config File
*/
int _mice_mod_rct_dummy_HandleConfFile(char *cConfFile)
{
  return(0);
}
```

Please avoid using this module because it may cause security problems. Just keep it as an example.

Chapter 4

Structures, Protocols and Interfaces

This chapter is reserved to describe all the different API and protocol structures used by the components of M-ICE.

4.1 Data Forwarder

The `DataForwarder` reads the raw log-data directly from syslog-like log-files or from device-files. Early testing of monitoring device-files revealed some misfunctions that I never fixed and never tested it again. The reason for it was that there are often programs like `syslogd` or `auditd` [1] that read, reformat and save the data to the filesystem. This fact puts alot of work away from us, but has the **extremly bad** side-effect that we do not read the data directly from the trusted [2] Kernel [8] by the trusted IDS. We need to expand our sphere of trust and make some assumptions to avoid the uselessness of our design [3]. In short we trust the Kernel, some system applications and the root user. For a more comprehensive list take a look at the Admin Guide.

Ok to come back to what is important at the moment, the `DataForwarder` reads ASCII data line-by-line from the files. It was never tested with binary data.

Every module that is used by the `DataForwarder` has to provide a `init`, `func` and `close` function. The function-name is assembled by the following pattern `<module name>_LTX_{init,func,close}`, like `mice_mod_flt_regex_LTX_init`.

To not unnecessarily blow things up I will just give some examples in the next subsections. Please be indulgent.

[1] read the LAuS documentation for more informations

[2] let us not argue about this, we need to make assumptions to avoid unwanted delays of our work

[3] you will find the same assumptions in security-certification levels of the USA government, like EAL3 of the Common Criteria [7]

4.1.1 Filter Module

```
int mice_mod_flt_regex_LTX_init(char *ConfFile);
int mice_mod_flt_regex_LTX_func(char *cLogData, size_t LogDataLen);
int mice_mod_flt_regex_LTX_close(void)
```

4.1.2 Pseudonymizer Module

```
int mice_mod_psd_empty_LTX_init(char *cConfFile);
int mice_mod_psd_empty_LTX_func(char *cLogData, size_t LogDataLen);
int mice_mod_psd_empty_LTX_close(void);
```

4.1.3 Format Module

```
int mice_mod_fmt_simple_LTX_init(char *ConfFile);
int mice_mod_fmt_simple_LTX_func(LogFormat *LogEntry, char *cLogData,
                                                      size_t LogDataLen);
int mice_mod_fmt_simple_LTX_close(void);
```

The `LogFormat` buffer has the following structure:

```
typedef struct
{
  char    cHost     [MAX_HOST]      __attribute__ ((packed));
  char    cDomain   [MAX_DOMAIN]    __attribute__ ((packed));
  char    cIP       [MAX_IP]        __attribute__ ((packed));
  char    cOSystem  [MAX_OS]        __attribute__ ((packed));
  char    cRelease  [MAX_RELEASE]   __attribute__ ((packed));
  char    cVersion  [MAX_VERSION]   __attribute__ ((packed));
  char    cDate     [MAX_DATE]      __attribute__ ((packed));
  char    cTime     [MAX_TIME]      __attribute__ ((packed));
  char    cLogLine  [MAX_DATA]      __attribute__ ((packed));
  u_short sChkSum                   __attribute__ ((packed));
} LogFormat;
```

4.1.4 Network Protocol

The `DataForwarder` sends one TCP packet per log-entry to the further components without building another, more abstract, protocol-layer on top of TCP. The packet is structured like the following.

```
typedef struct
{
  u_int       IVLen                                 __attribute__ ((packed));
```

```
  char        IV[16]                              __attribute__ ((packed));
  u_int       CipherTextLen                       __attribute__ ((packed));
  char        cCipherText[1*sizeof(LogFormat)]    __attribute__ ((packed));
} CipherMsg;
```

Please note that I want to avoid the IV to be send in plaintext along the message in future by deriving it from the secret key. Further a `IVLen` of 0 indicates a plaintext message.

4.2 Raw Log Database

The API for the different decoding and the post-processing modules look always the same.

4.2.1 Decoding Module

```
size_t mice_mod_dec_logformat_twofish_LTX_init(char *ConfFile);
char  *mice_mod_dec_logformat_twofish_LTX_func(char *cData, size_t DataSize);
int    mice_mod_dec_logformat_twofish_LTX_close(void);
```

4.2.2 Postprocessing Module

```
size_t mice_mod_pop_mysql_LTX_init(char *ConfFile);
int    mice_mod_pop_mysql_LTX_func(char *cData, size_t DataSize);
int    mice_mod_pop_mysql_LTX_close(void);
```

4.3 Analysis Agent

The API for the different decoding and the post-processing modules look always the same.

4.3.1 Decoding Module

```
size_t mice_mod_dec_logformat_twofish_LTX_init(char *ConfFile);
char  *mice_mod_dec_logformat_twofish_LTX_func(char *cData, size_t DataSize);
int    mice_mod_dec_logformat_twofish_LTX_close(void);
```

4.3.2 Postprocessing Module

```
size_t mice_mod_pop_aa_regex_LTX_init(char *ConfFile);
int    mice_mod_pop_aa_regex_LTX_func(char *cData, size_t DataSize);
int    mice_mod_pop_aa_regex_LTX_close(void);
```

4.4 Management Host

The API for the decoding and the post-processing modules look always the same.

4.4.1 Decoding Module

```
size_t mice_mod_dec_idmef_twofish_LTX_init(char *ConfFile);
char  *mice_mod_dec_idmef_twofish_LTX_func(char *cData, size_t DataSize);
int    mice_mod_dec_idmef_twofish_LTX_close(void);
```

4.4.2 Postprocessing Module

```
size_t mice_mod_pop_act_generic_LTX_init(char *cConfFile);
int    mice_mod_pop_act_generic_LTX_func(char *cData, size_t DataSize);
int    mice_mod_pop_act_generic_LTX_close(void)
```

4.5 Reaction Agents

The API for the reaction-agents always looks the same. All agent process the following type of message:

```
typedef struct
{
  char    cIdmefMsg[MAX_IDMEFMSGSIZE+1]          __attribute__ ((packed));
  char    cAlertID[RIDMSG_MAX_ALERTID+1]         __attribute__ ((packed));
  char    cAlertIDDesc[RIDMSG_MAX_ALERTDESC+1]   __attribute__ ((packed));
  int     iRID;
} RIDMsgFormat;
```

4.5.1 Save to File

Just writes the IDMEF message to a file.

4.5.2 Send to Syslog

Just sends some of the data to `syslogd`.

4.5.3 Send to Database

Just the MySQL query.

```
char *cQueryFormat = "INSERT INTO alert_entry (alertid, alertdesc, classification,
                      date, time, analyzerid, source_address, source_user,
```

```
      source_process, source_service, target_address, target_user,
      target_process, target_service, idmefmsg, signature)
                      VALUES ('%s','%s','%s','%s','%s','%s','%s','%s',
      '%s','%s','%s','%s','%s','%s','%s','%s')";
```

4.5.4 Countermeasure

The reaction-message, that was triggered by a specific alert, is send to the client's `ReactionDaemon`.

```
typedef struct
{
  char      alert_id[RIDMSG_MAX_ALERTID+1];
  u_int     reaction_id;
  uid_t     uid_for_exec;
  gid_t     gid_for_exec;
  u_int     function_id;
  u_int     num_of_args;
  char      arg_fmt_string[MAX_ARGSTRG_SIZE+1]  __attribute__ ((packed));
  char      arg_fmt_param[MAX_FMTSTRG_SIZE+1]   __attribute__ ((packed));
} stExecMsg;

typedef struct
{
  u_int     show;
} stShowMsg;

typedef struct
{
  u_int     function_id;
} stCheckMsg;

typedef struct
{
  int     ret_val;
} stRetvalMsg;

typedef struct
{
  struct
  {
    u_int   uiID;
```

```
    char    *cModName[MAX_MODNAME+1]  __attribute__ ((packed));
  } Function[MAX_FUNCID]              __attribute__ ((packed));
} stAllMsg;

typedef struct
{
  u_int     supported;
} stSupportedMsg;


typedef struct
{
  short     sChkSum;
  time_t    Timestamp;

  u_int     Mode;

  union
  {
    stExecMsg       Exec;
    stShowMsg       Show;
    stCheckMsg      Check;
    stRetvalMsg     Retval;
    stAllMsg        All;
    stSupportedMsg  Supported;
  } ModeData;

} stReactionMsg;

typedef struct
{
  u_int       IVLen                                 __attribute__ ((packed));
  char        IV[16]                                __attribute__ ((packed));
  u_int       CipherTextLen                         __attribute__ ((packed));
  char        cCipherText[sizeof(stReactionMsg)]    __attribute__ ((packed));
} stCipherRctMsg;
```

The enciphered reaction-message looks like the logformat-message send from the `DataForwarder`.

4.6 Reaction Daemon

The `ReactionDaemon` just reads a reaction-messages from the network and forwards it to the reaction-module. The API for the reaction modules look always the same.

4.6.1 Reaction Module

```
size_t mice_mod_rct_dummy_LTX_init(char *ConfFile);
int mice_mod_rct_dummy_LTX_func(char *cArg, size_t ArgLen);
int mice_mod_rct_dummy_LTX_close(void);
```

Chapter 5

ToDo

- release

Appendix A

Abbreviations

DAC Discretionary Access Control (all files are protected by DAC rules)

DoS Denial–of–Service

FIFO First In, First Out; Named Pipe; local Interprocess Communication

GNU GNU's Not Unix!, Projekt of the *Free Software Foundation*

GUI Graphical User Interface

IDMEF Intrusion Detection Message Exchange Format

IDS Instrusion Detection System

IP Internet Protocol, s. RFC–791 [3]

LAuS Linux Audit-Subsystem

LKM Loadable Kernel Modul

PAM Pluggable Authentication Module

SO Security Officer

SQL Structured Query Language

SSL Secure Socket Layer, Encryption on presentationlayer

Syslog native Unix Logging System

TCP Transmission Control Protocol, s. RFC–793 [3]

UDP User Datagram Protocol, s. RFC–768 [3]

UML Unified Modeling Language

XML Extensible Markup Language

Appendix B

List of Figures

Appendix C

Bibliography

[1] D. Curry, H. Debar; Intrusion Detection Message Exchange Format — Data Model and Extensible Markup Language (XML) Document Type Definition; IDWG; February 2002

[2] LibIDMEF, http://www.silicondefense.com/idwg/libidmef/index.htm

[3] RFC Datenbank, http://www.rfc-editor.org/

[4] GNU Free Documentation Licence, http://www.gnu.org/copyleft/fdl.html

[5] M-ICE project at Sourceforge, http://sourceforge.net/projects/m-ice/

[6] MySQL, http://www.mysql.com

[7] CAPP Version 1d, http://www.radium.ncsc.mil/tpep/library/protection_profiles/CAPP-

[8] Paul E. Proctor; The practical Intrusion Detection Handbook; Prentice Hall

[9] LibTool; http://www.gnu.org/directory/libtool.html

[10] CVS Book; http://cvsbook.red-bean.com/

[11] Autotools Book; http://sources.redhat.com/autobook/