Developing additional EAP methods for XSupplicant

Written By, Chris Hessing (chris@open1x.org)

Target Audience

This document is targeted at developers that wish to add additional EAP methods to XSupplicant. This document assumes that the reader has a basic understanding of 802.1X, and 802.11. This document also assumes that the reader has a deep understanding of programming in the C programming language, and of the Extensible Authentication Protocol (EAP) standards.

Introduction

One of the design goals of the current XSupplicant code base is that adding new EAP methods should be as easy, yet powerful, as possible. Another set of design goals that anyone developing a new EAP method should keep in mind is that XSupplicant tries to stay as true to the standards as possible. With this in mind, it is also important to realize that different people will interperate standards in different ways. To this end, any EAP implementation should strive to maintain maximum compatibility with existing server implementations. In the event that the common server implementations differ significantly from the documented standards, the developer should provide a configuration option that allows for strict compliance with the standard, and have the default behavior be whatever is most likely to work.

An example of allowing a strict standards compliance mode can be found in the existing PEAP version 1 code. According to the standards, the keying material should be generated using the string "client PEAP encryption". However, the common implementation is to use "client EAP encryption". As a result, the default behavior of the supplicant is to use "client EAP encryption", but a "Proper_PEAP_V1_Keying" option is available in the configuration file to enable a mode that follows the text of the standard.

File System Layout

When working in the engine component of the supplicant, the file system layout should be as follows :

```
\xsupplicant
        \src
                \eap_types
                        \<method name>
```

Where <method name> is the name of the EAP method you plan to implement, without the preceding "EAP" potion. (i.e. If you were implementing EAP-MD5, the directory name should be simply "md5".)

Everything that is specific to your EAP method should be kept in this directory. Also, while not required, it is a good idea to name your files eap<method_name>_*.c/h. (i.e. The basic PEAP implementation would be named eappeap.c/h. If a support file were needed to work through the development, it would be named eappeap_support.c/h. The exception to this is if a file may be used in multiple EAP methods such as the TLS handler code in EAP-TLS. In this case, the filename can be a free-form string that is meaningful. **NOTE :** The filenames are important. The name of a file ties directly in to the style that should be followed during all supplicant related development. Please see the style guide for more information.

There are also some functions that may be useful across all, or almost all EAP methods. In the event that this is the case, those functions should be stored in code located at the xsupplicant\src\eap_types level of the filesystem. At the time of writing, there is currently an eap_type_common.c/h file. It is probably a good idea to include this header in any EAP implementation. Before implementing any new support code at this level, please verify that it isn't already included in eap_type_common.c/h.

Because EAP methods are inherently engine only, there should be no UI development needed when implementing the EAP method. **NOTE : There will probably be EAP method configuration development that will need to be done in the UI. However, that is beyond the scope of this document. Please see the document on EAP method configuration for information on developing a configuration handler for the UI.**

Adding your files to the build system

Because XSupplicant is a cross platform supplicant, it is difficult to be able to add something to the build system and have it work across all of the target platforms. Because of this, a complete description of adding your files to the build system is beyond the scope of this document.

If you are developing on Windows, you should add your headers to the "Header Files" section and your source files to the "Source Files" section of the XSupplicant project in Visual Studio.

If you are developing on Linux, you should add both your headers and your source files to the "xsupplicant_SOURCES" section of the Makefile.am. You should be able to use any text editor that your are comfortable with when editing this file.

Please remember that when submitting new EAP methods to the supplicant, you will be required to identify any new files that are added, and indicate which platforms they were developed on, and which platforms they were tested on. This will allow other developers to assist in getting your code building on platforms other than the one you developed on.

Beginning your EAP method implementation

Once you have made it this far, you should be ready to being your implementation. Your first stop should be the src\eap_sm.c file. Here you will need to add the function pointers that the supplicant will use to call in to your EAP method. The majority of the function calls that you will need to provide map directly to function calls that the EAP RFC calls out. However, because of some implementation details, there are a few other functions that are also required.

Note: EAP-MD5 is probably one of the most simple EAP methods to implement. As a result, it is probably also some of the most easy code to reference while doing your implementation. If you have any questions about how to properly implement your EAP method, it is suggested you look at the EAP-MD5 code first, and then ask on the mailing list(s).

The first thing you need to do is stub in the functions that you need for your EAP method. These stubs should be put in the source files that you will use for your EAP method development. (i.e. If you were developing EAP-MD5, you would put these stubs in the src\eap_types\md5\eapmd5.c/h files.)

In order for your implementation to work, you will need to provide implementations for the different members of the rfc4137_eap_handler structure. This structure contains everything the supplicant needs to know in order to properly call your EAP method. It is also important to note that the structure contains both function pointers, and regular variables. Where function pointers are in use, there is a pointer to an abstracted structure that should contain the information that most EAP methods will need to complete their implementation. If the information you need isn't provided in the existing structure, you have two choices on how to proceed.

1. You can add the needed information to the structure. However, you need to keep a few details in mind when choosing to go this route.
   - The information included in this structure MUST be generic in nature. The consumer of the structures are generally not aware if they are being run in a tunneled method. Any information that is included in this structure should assume that it may be inside of an arbitrary number of tunnels. Because of this, it is generally a good idea to ask on the mailing list before taking this route.
   - Any changes made will need to be propagated through all other EAP methods. In general, this shouldn't be much of an issue. However, it is something worth considering and looking in to. A simple change to make your implementation easier, may cause a lot of headaches for other developers.
   - Any changes made will also require documentation updates. (i.e. You MUST update this document to reflect the changes. Failure to update this document along with your changes may result in your changes being rejected from the source tree!)

2. If your method is a corner case that requires knowing something that is only provided in the context, you method can make use of the event_core_get_active_ctx() function. However, doing this has it's own risks. Because your EAP method won't know how deeply it is tunneled, it will be difficult for your EAP method to know what configuration information to read. So, this method should ONLY be used if you are looking for top level context information. (Such as an interface name, full frame data, source MAC address, destination MAC address, etc.)

Before getting in to the structure members that need to be implemented, it makes sense to look a little closer at the eap_type_data structures as members of the rfc4137_eap_handler may need to change values in the eap_type_data structure to signal their status.

The eap_type_data structure is as follows :

eap_conf_data – This is a pointer to a memory blob that will contain the configuration data for your EAP method. If this is a tunneled EAP method, it will point to the proper configuration data for the level of tunneling that your EAP method is currently at. (i.e. You should never need to worry about reading in to a deeper phase of data to get the configuration information for you EAP method.) ***Note:*** You should NEVER modify the values of this configuration structure. Any changes that you make will be changed globally!

eap_data – This pointer is for your EAP method to use to store information that will survive from one call in to the next. Your EAP method should allocate this pointer if it chooses to use it, and deallocate it when it is finished. You may type cast this pointer to a structure that your EAP method uses to track various bits of data. However, the supplicant engine will ALWAYS view this as nothing more than a memory blob. Because of this, the lower layers of the supplicant will be unable to use any data

contained in this blob.

methodState – This is an unsigned 8 bit number that indicates the state of the EAP method as it is running. The EAP method should ALWAYS be sure that this value is set to something when it terminates one of it's primary EAP functions. The lower layers of the supplicant use this value to determine if the EAP method needs to be called again to continue the authentication, or if it is done. This variable should ONLY be set to one of the following values :

- DONE : Your EAP method should set this value when it has nothing more to do in the conversation. When the lower layers of the supplicant see that this value is set, they will look at the value of decision (below) to determine if the EAP method believes that the authentication should be successful or not.
- INIT : Your EAP method shouldn't ever need to set this value. However, when the lower layers of the supplicant want to let your EAP method know that it should start fresh, it will set the methodState to INIT. Keep in mind that the lower layers of the supplicant may decide to send you an INIT at any time. Because of this, you should always check to see if the value is INIT. If it is, you should clean up anything from the last attempted authentication, and reset for a new attempt.
- CONT : Your EAP method should set methodState to CONT when it knows that it has more work to do in the authentication. Keep in mind that the CONT state indicates that your EAP method has more work to do no matter what the other end of the connection thinks. Because of this, CONT should be used when your method is expecting something like an ACK from a fragment.
- MAY_CONT : Your EAP method should set methodState to MAY_CONT when it may have more work to do in the authentication. MAY_CONT state indicates to the lower layers of the supplicant that your EAP method is in a state where the other side could decide to accept, fail, or contine the authentication. MAY_CONT state should be used any time when you aren't 100% sure that you will continue, or be DONE.

decision – This is an unsigned 8 bit number that indicates to the supplicant's lower layers if the EAP method believes the authentication should be successful. If your EAP method doesn't know enough about the authentication to be sure, it should leave the decision set as NONE. If you reach a situation where your EAP method cannot continue for some reason, you should set the methodState to DONE, and decision to EAP_FAIL. This will cause the supplicant lower layers to abort the authentication attempt, and discard the existing information in the buffer. If your EAP method believes that the authentication should be successful, it should set this value to either UNCOND_SUCC or COND_SUCC. The difference between the two is how optimistic the EAP method is about it's chances of success. UNCOND_SUCC means that the EAP method believes it will succeed no matter what. COND_SUCC means that the EAP method believes it will succeed as long as the other end of the conversation agrees. In general, you want to use COND_SUCC unless your EAP method receieves some indication that the server will accept the authentication.

eapReqData – This is a pointer to the area of the received frame that contains the EAP data. The length of this data is specified by the length value that is included in the EAP header. So, in order to determine how much data is there, you should reference the length value provided by EAP. To make things easier, the eap_type_common.c/h files contain a function that you can use to return the length value. **Note:** Your EAP method should NEVER modify the contents of this variable! Should you need to do any operations on the data that is included in it, you should first make a copy.

ignore – This value indicates to the lower layers of the supplicant if the EAP packet that is currently being looked at should be discarded and ignored. When this is set to TRUE, the packet will be ignored. When set to FALSE the lower layers will believe that the EAP method has processed the packet and handled it properly.

eapKeyAvailable – In general, your EAP method shouldn't care about the value of this variable. This variable indicates if the lower layer(s) of the supplicant believe that they is key data currently available.

altAccept – This value should be set to TRUE if the EAP method provides some method of signaling internally that an authentication has succeeded. As an example, as part of EAP-MSCHAPv2, the server will send a success with various pieces of information. This success will not be a standard EAP-SUCCESS message, but will be identified as a normal EAP-MSCHAPv2 packet. When EAP-MSCHAPv2 gets it's encoded success, it should set this value to TRUE.

altReject – This value should be set to TRUE if the EAP method provides some method of signaling internally that an authentication has failed. As in the altAccept case, EAP-MSCHAPv2 provides an internal indication of a failure. If the interal failure indication is signaled, then EAP-MSCHAPv2 should set this to TRUE.

ident – This is a pointer to a string that contains the identity value that would have been used in response to any EAP-RequestID messages the supplicant would have gotten. Unless your EAP method as some reason to do otherwise, this is the value that should be used when calculating any hashes that involve the username.

Now that we have outlined the eap_type_data structure, we should have enough information to get in to how to put an EAP method together. When implementing an EAP method, the following structure members (members of the rfc4137_eap_handler struct) MUST be popluated :

eap_type_handler - This should be a #define(d) value that references the EAP number that this EAP method binds to. You should put the #define in the xsupconfig.h file that is located in xsupplicant\lib\libxsupconfig directory. As an example, if you were implementing EAP-MD5, you would want to add :

```
#define EAP_TYPE_MD5   4
```

to the xsupconfig.h file. Then, any time you wish to reference the EAP number for the MD5 method, you should use this define.

eapname – This should be defined as a string that identifies the EAP method that you are implementing. This string can be defined directly in the structure for the EAP method. Again, using EAP-MD5 as our example method, this string would be defined as “EAP-MD5”. There is no need to define this value as a #define, or specifically allocate memory for it. You can just define the string inside of the structure member of the array.

eap_check(eap_type_data *) – This is a pointer to a function that is called each time a frame is received for the specified EAP method. The purpose for this function is outlined in some detail in the EAP RFC. However, in a nutshell this function should validate the input frame, and set flags in the eap_type_data structure that indicate if everything is in order to process this frame and build a response. There are

any number of things that can be checked in this function, but a bare minimum would be to validate the structure of the data being passed in, and verify that any numeric values are within reasonable ranges. If something fails to check out, you can attempt to set the proper eap_type_data member variables to indicate failure. However, it is suggested that you just use the eap_type_common_fail() function that is provided in the eap_type_common.c/h files. Using the eap_type_common_fail() function will ensure that your method behaves properly in any future changes.

eap_process(eap_type_data *) - This is a pointer to a function that will be called to process the information that is included in the EAP method. This function is also defined by the EAP RFC, and the RFC should be considered to be the authoritative source on information about this function. However, in a nutshell this function should read the data provided in the EAP method, and do any processing that is needed to create a response. Like the eap_check() function, should your EAP method reach a failure case, you should use the eap_type_common_fail() function to indicate failure.

eap_buildResp(eap_type_data *) - This is a pointer to a function that will generate an EAP packet in response to the frame that was processed in the eap_process() call. This function should return a pointer to the response EAP packet or, if a failure case is reached, NULL. In the event a failure case is reached, this function should use the eap_type_common_fail() function call to indicate failure.

eap_isKeyAvailable(eap_type_data *) - This function should return TRUE of FALSE to indicate if keying material is currently available to the lower layers of the supplicant. If this function returns TRUE then the eap_getKey() call below needs to return valid key data.

eap_getKey(eap_type_data *) - This function should return a pointer to a validate byte array that contains the key generating material that was obtained by the EAP method during the authentication. In general, the lower layers of the supplicant are expecting that 32 bytes of keying material are returned. If your method returns more or less than this, please see the note on the eap_getKeyLen() function below.

eap_getKeyLen(eap_type_data *) - This function should return a number that indicates the size of keying material to be returned. This is one of two functions in the structure that are not defined by the EAP RFC. Because of this, it should be assumed that this document is the authoritative source of information on this function. By default, the supplicant knows how to deal with keys that are COMMON_KEY_LEN in size. (COMMON_KEY_LEN is defined in eap_type_common.h, and is currently 32.) In the vast majority of cases, you should define this value to point to the eap_type_common_get_common_key_len() function. This function will help ensure that your method does the right thing if lower layer changes are needed in the future. ***Note:*** If your EAP method doesn't return any keying material, you should use the eap_type_common_get_zero_len() function. ***Note:*** If your EAP method returns a value other than COMMON_KEY_LEN, you will need to make some changes to the lower layers of the supplicant. (These changes are FAR beyond the scope of any of the development documents. It is suggested you contact the list for help.) Currently, the only method that uses a key length that isn't 32 is LEAP. So, should you believe that your EAP method is different, please go back and double check.

eap_deinit(eap_type_data *) - This function should clean up any memory that your EAP method may have used. In general, this function is a good place to clean up the eapData member of the eap_type_data structure. Keep in mind that any memory that isn't cleaned up here will be leaked!

Now that all of your EAP functions are stubbed in, it is time to add them to the EAP method array that is part of xsupplicant\src\eap_sm.c. Although I have used EAP-MD5 as an example throughout this document so far, I am going to switch to using EAP-SIM here, since the EAP-SIM definition illustrates some important things to consider where doing an EAP method implementation.

The EAP-SIM member of the EAP type array looks like this :

```
#ifdef EAP_SIM_ENABLE
  {EAP_TYPE_SIM, "EAP-SIM", eapsim_check, eapsim_process, eapsim_buildResp,
   eapsim_isKeyAvailable, eapsim_getKey, eap_type_common_get_common_key_len,
   eapsim_deinit},
#endif
```

The first thing to notice about this definition is that it is enclosed in an #ifdef. If your EAP method requires libraries outside of the normal set of libraries required to build the supplicant, you should enclose your method in similar #ifdef settings, and document the values needed to enable your EAP method. The value that is used to indicate that your code should be built should also enclose all off the code for your EAP method implementation. As an example, please see the eapsim.c file. (Note : You should enclose ALL of your code in ALL of your EAP method implementation files in the #idefs!)

After the #ifdefs, you will notice that the array element contains the structure for the rfc4137_eap_handler struct defined above. If you populate this according to the information provided for each structure member, you should have almost everything you need to get your EAP method running.

The last thing that you should need to do is make sure that the header files for you EAP method are included in the eap_sm.c file so that the function names defined in the array can be resolved at compile time.

Providing log information, and debugging information

As you work on your EAP method, you will probably want to provide different types of log information, and different types of debug information. The xsup_debug.c/h files should contain everything that you need to effectively add both logging and debugging information to your project. The design goal of the xsup_debug functions were to make the difference between a logging a debugging function as small as possible so that it would be easy for a developer to change their mind on what information the user should see, and what information they shouldn't.

There are a few key functions that you will usually use to provide both logging a debugging information :

`void debug_printf(uint32_t, char *, ...);` - This is the most basic replacement for the printf() function that is provided in C. The prototype used to call this function is nearly the same as well. The key different between the way this function is called, and the way the normal printf() function is called, is that debug_printf() expects the first parameter to be a uint32_t that indicates the debug level(s) that this should be displayed at. The number of debug levels that are available is somewhat dynamic. However,

the ones that an EAP method developer would want to use don't change much. They are DEBUG_NORMAL, DEBUG_VERBOSE, and DEBUG_AUTHTYPES. DEBUG_NORMAL is the most basic debug level. When DEBUG_NORMAL is used, the log information that is provided by this call will ALWAYS be logged to the appropirate logging mechanisams. Because it is always logged, any DEBUG_NORMAL messages should be written in a way that any user should understand. DEBUG_VERBOSE messages will only be displayed when the user has selected verbose mode. (This is usually selected through a UI that is controlling the supplicant engine.) Verbose messages will also usually be seen by a user, so they can be more technical that a NORMAL level debug message, but should still be something that a user could understand. DEBUG_AUTHTYPES are debug messages that are generated by authentication methods. This type of debug message will only ever be logged. The only way a user will see this is to open up any log files (or run the process in foreground mode). These messages can be as technical as the developer needs them to be in order to get the job done.

void debug_printf_nl(uint32_t, char *, ...); - This call is the same as the debug_printf() call above. The main difference is that the debug_printf() function will always display a "label" at the beginning of the line to indicate the logging level that generated the message. This version of the call will not generate the label. In general, you should use debug_printf() unless you have a good reason to use this version.

void debug_hex_printf(uint32_t, uint8_t *, int); - This function call will dump an array of data to an ASCII representation of the hexidecimal that the array contains. Like the other debug_* functions, the first parameter is the debug level to display this information at. In general, you shouldn't use this function to display DEBUG_NORMAL level messages. (Most users won't understand what the hex means!) The second parameter is a pointer to the data you want to display, the third is the number of bytes to display from the data provided in parameter 2. Like other printf calls, this function will display all of the data on a single line. As such, it is probably not what you want to use if you are going to display a large amount of data. (Usually you should use this function with anything of a length up to about 32 bytes.)

void debug_hex_dump(uint32_t, uint8_t *, int); - This function call serves a similar function to the debug_hex_printf() function call. The key difference is this version is built to handle large amounts of data, and provides prettier output for the data. This hex dump call will give you three columns of output. In the first column is the offset of the data in to the data blob. (The offset is a 12 bit number, so it will roll over for really large data sets!) The second column contains the ASCII representation of the hexidecimal data. The third column will display the ASCII character for each byte of data, or a period if the character is non-printable. The end result of this style of output is something that resembles the output of a sniffer program such as Wireshark.

Conclusion

Hopefully the information provided in this document is enough to allow you to develop a new EAP method for XSupplicant. As you are developing your method, please remember that XSupplicant only exists because of the contributions of the community around it. We understand that not all development effort can be returned to the community, but we do apprciate anything that can be provided back to the community. This may include bug fixes, or suggestions to improve different aspects of the behavior of the supplicant.

If you would like to contribute code or suggestions, but don't want to join the mailing list to do so, or don't want your e-mail address identified, please feel free to e-mail me directly. (My e-mail address is at the top of this document.)