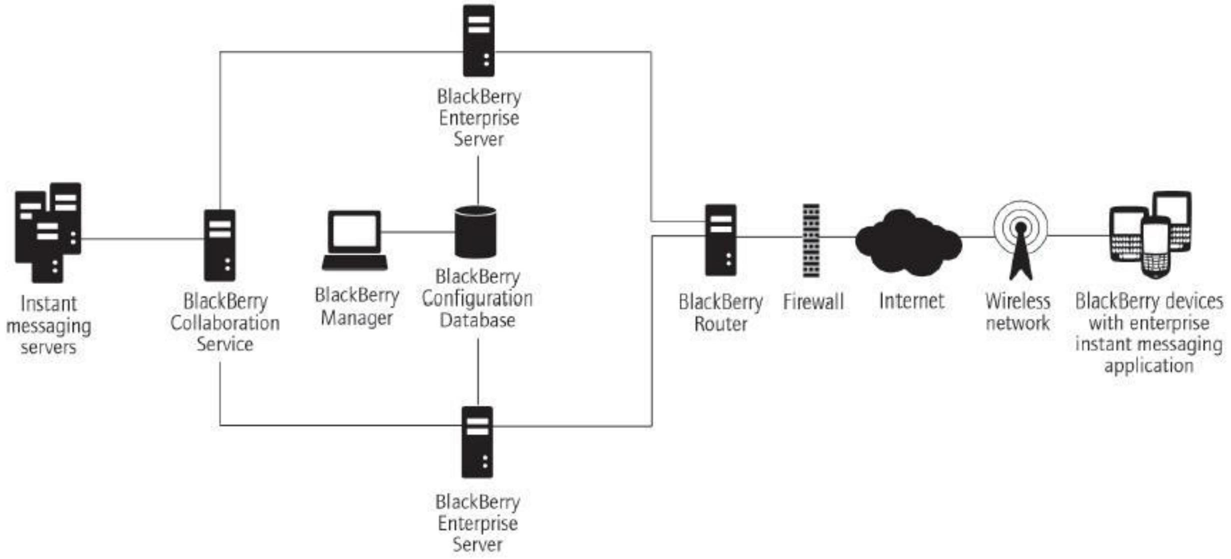
Research Month Summary

Overview

BES is used to provide mobile devices users access to enterprise email and messaging services, such as Exchange and Office Communicator. Versions for Lotus Domino and Novell GroupWise are also available, but the focus of my research was on the Exchange version. However, the majority of the code is likely shared between all versions. Any organization that allows corporate users to use a BlackBerry to check email will have a BES server of some sort, depending on the underlying messaging system. The architecture of the BES server looks like this:


BlackBerry Enterprise Server
Instant messaging servers
BlackBerry Collaboration Service
BlackBerry Manager
BlackBerry Configuration Database
BlackBerry Router
Firewall
Internet
Wireless network
BlackBerry devices with enterprise instant messaging application
BlackBerry Enterprise Server

**Picture taken from the BES documentation.*

The BES server resides behind the Firewall, and is not directly accessible from the Internet, or at least it shouldn't be. The vector for attacking the server is through corporate end users that have BlackBerries. If an end user can be convinced into opening up an attachment we send them, then it is possible to cause the BES server to convert the attachment into a format viewable on the end user's BlackBerry device. It is important to note that the targeted user must attempt to view the attachment in order for the conversion process to occur. The parsing of the various attachment types opens up a large attack surface to the remote attacker. All that is needed is an email address within the target enterprise, and a user that can be socially engineered into opening an attachment.

The purpose of this document is to aid others in their efforts to uncover vulnerabilities in BES by describing the various parts of BES and how they operate, with an emphasis on those parts most likely to contain remotely exploitable vulnerabilities

Basics

The first step towards owning the BES server is installing it. There are two choices here, a full installation of the server and the BlackBerry phone emulator, or a minimal installation of just the attachment server. The first choice requires installing Exchange 2003, setting up a domain, installing the full BES system, and finally installing a BlackBerry phone emulator and connecting it to the BES server. This task took several days the first time it was performed. This choice most closely mirrors the actual real world environment, and is what I chose to do due to lack of further knowledge of how everything actually works. It's time consuming and boring. Also, it is unneeded. Instead of doing all of this, you can simply install the attachment server component. RIM chose to make BES very modular, and it is composed of a handful of different components (services). Nearly all of these components can be installed on different boxes and then linked together. The attachment server can be installed on any Windows platform, and comes with a handy program that can be used to test it (convert attachments). For doing research, this is the simplest choice, and as far as I can tell, it behaves exactly as a full and properly configured environment would.

Assuming you have it installed, you'll find the following directory in the installation directory:

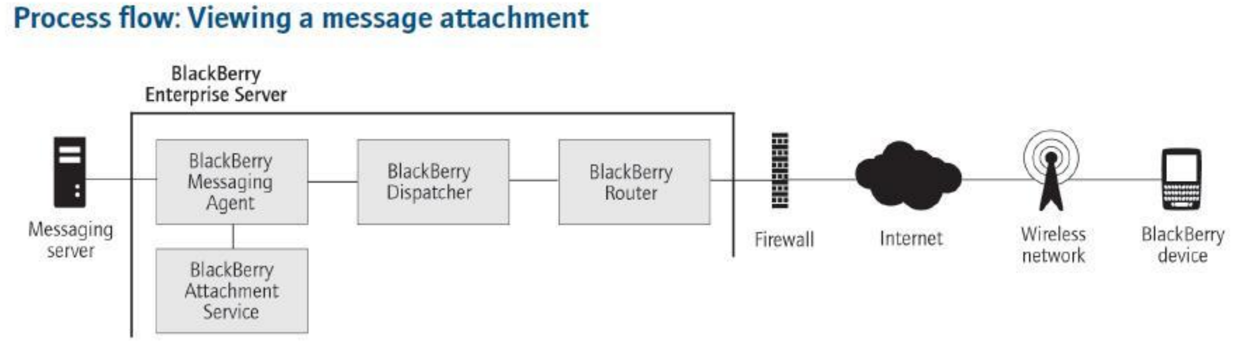
C:\Program Files\Research In Motion\BlackBerry Enterprise Server\AttachServer\BBDistiller

This directory is filled with a bunch of DLLs in the form of BBDM_fileformat.dll. These are all of the file parsing and converting DLLs. Their names describe what file formats they parse (not all of them actually parse the formats though, more on that later). These DLLs are all part of a document conversion library called AirDoc, originally distributed by a company called Arizan that RIM later purchased. This library is a set of COM libraries made for parsing various file formats and converting them into a format that is easy to render on mobile devices. At one time this company had a public site, and it contained a developer's section that probably had documentation for the AirDoc library. However, after searching I was unable to find any piece of documentation still around.

The AirDoc library is a set of COM components, and BES as whole makes heavy use of COM. Nearly all of the server components talk to each other using COM libraries. This can be a pain, but it also extremely useful; the code is all C++, makes heavy use of OOP, and has lots of RTTI information due to dynamic_cast calls. All of the DLLs mentioned above have RTTI data, and it can be recovered using igorsk's RTTI scripts from openrce. This information was indispensable when attempting to understand how things work.

Another helpful COM-provided bit of information is typelibs for all of the attachment processing DLLs, and many of the communication DLLs as well. To view this info, fire up OLEView, browse to the 'All Objects' node, and then look for anything that starts with 'Arzn'. For example, ArznFI_MP3 class has the typelib information for the MP3 parser. There are 50+ different COM classes, and nearly all of them have typelib info. This lets one view the MIDL description files for them, which is a big help when used in combination with the RTTI scripts. It's relatively easy to find the file parsing code with this information in hand.

How it works



Process flow: Viewing a message attachment

**Picture taken from the BES documentation.*

The following is taken from the BES documentation:

1. A user receives a message with an attachment on a BlackBerry® device.
2. The BlackBerry Messaging Agent verifies that the format of the attachment is valid for conversion.
3. The user clicks the Open Attachment menu item to view the attachment on the BlackBerry device.
4. The attachment viewer sends the request to the BlackBerry Messaging Agent, which connects to the BlackBerry Attachment Service over port 1900.
5. The BlackBerry Attachment Service retrieves the attachment in binary format from the user’s message store using the BlackBerry Messaging Agent link to the messaging server. The BlackBerry Attachment Service distills the attachment and extracts the content, layout, appearance, and navigation information from it.
6. The BlackBerry Attachment Service organizes, stores, and links the information in a proprietary DOM in a binary XML style. The BlackBerry Attachment Service formats the attachment for the BlackBerry device and converts it to UCS format.
7. The BlackBerry Attachment Service sends the UCS data to the BlackBerry Messaging Agent using a TCP/IP connection over port 1900.
8. The BlackBerry Messaging Agent sends the converted attachment to the BlackBerry Dispatcher.
9. The BlackBerry Dispatcher compresses the first portion of the attachment, encrypts it with the master encryption key of the BlackBerry device, and sends the first portion of the attachment to the BlackBerry Router.
10. The BlackBerry Router sends the first portion of the attachment to the wireless network over port 3101, which verifies that the PIN belongs to a valid BlackBerry device that is registered with the wireless network.
11. The wireless network delivers the attachment to the BlackBerry device. The BlackBerry device sends a delivery confirmation to the BlackBerry Dispatcher, which sends it to the BlackBerry Messaging Agent. If the BlackBerry® Enterprise Server does not receive a delivery confirmation within 4 hours, it sends the attachment data to the wireless network again.
12. The BlackBerry device uses its master encryption key to decrypt and decompress the attachment so that the user can view it.

13. The user views the attachment on the BlackBerry.

The specifics of step 5 and 6 are the primary points of interest, as this is where attacker supplied input in parsed and manipulated. The details of step 5 are as follows.

The first thing the Attachment Server (AS) does when it starts up is start X number of conversion processes. These processes run the BBConvert.exe application, located in the AttachServer directory. The number of conversion processes can be set using the supplied GUI configuration program, but it does not let you set it lower than 2. This can be annoying for debugging because you don't know which process will be converting your attachment. So, you can set the following registry key to make it start only a single conversion process:

```
HKLM\SYSTEM\ControlSet001\Services\BBAttachServer\Parameters\ProcessesNo
```

When the AS starts each process, it passes it a command line like this:

```
BBConvert.exe "\\.\pipe\Arzn_AdDocR1_ControlY" X Y
```

where Y is the control pipe #, and X is the document cache size. The first argument is a named pipe which is used as a command and control channel. It is a bi-directional pipe used by the AS and BBConvert processes for synchronizing such tasks as: starting a conversion process, sending the file data, receiving the converted file, and so on. The document cache is a cache used to store recently converted documents. This cache is maintained via MD5 sums of the file contents. This can present a problem when you want to test the server over and over with the same file. If the document is already in the cache, it will just serve it from there rather than reparse it. The simple way around this:

```
echo "A" >> test.doc
```

All of the file parsers I've looked at will let you append characters to the end and still reach the main parsing code.

After starting the conversion processes, the AS waits for an incoming conversion request on port 1900. When received, it chooses a slave process from its pool, and initiates the conversion commands on the control named pipe for the given slave process. The protocol for this pipe (and another pipe, the transmission pipe) is as follows:

Packet layout:

[DWORD request length (not inclusive of this field)]

[BYTE opcode]

[opcode dependent data, (len - 1 bytes)]

Opcodes:

server -> client:

C: 7 -> here comes the name of the transfer pipe

T: 3 -> xml header describing the file is coming

T: 6 -> here comes the file contents

client -> server:

C: 1 -> hello, i am here and ready for requests

C: 9 -> ready for file on transfer pipe

T: 5 -> give me the file contents

T: 0xa -> done with conversion

T: 0xb -> here is the converted file attachment

Notice the '**T**' and '**C**', this stands for transfer pipe and control pipe. Opcode 7 from the server is used to send the name of a transfer named pipe, which is named like "\\.\pipe\Arzn_AdDocR1_TransferX", where X is some number. The transfer pipe is used for sending and receiving the file contents, as well as the converted file. The server opens this endpoint, and the client connects to it. Some requests are sent down the control pipe, and others down the transfer. However, the transfer pipe closes after work is done, but the control pipe stays open for the duration of the process. Observing these calls can be done by breaking on ReadFile() and WriteFile().

Once the slave process receives the file contents, it loads up all of the COM libraries for parsing files. It calls the LoadDocument() function (this is in the MIDL interface description) on each COM interface until one succeeds. The recognition code for file formats is handled by a function call in the LoadDocument() function, that checks for magic bytes in the file to recognize its format. Once the conversion succeeds, the converted file (XML like format) is sent back across the transfer pipe to the AS.

To aid in testing purposes, I wrote an application called 'fake-pipe'. This application pretends to be the AS, and can be used to test the conversion process without running the AS service. It is located in the app/ directory of the research zip. It is run like this:

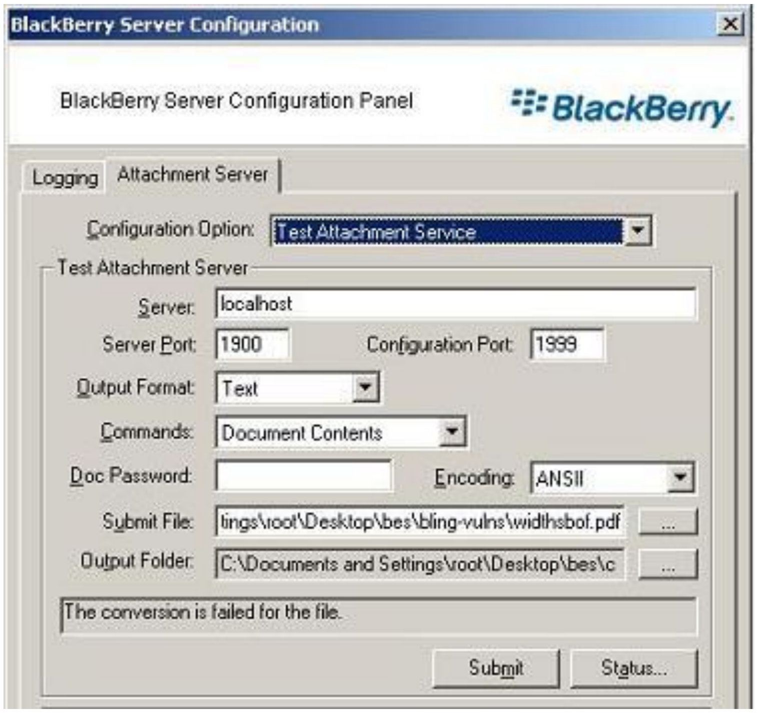
```
./fake-pipe.exe "file_name"
```

After it starts, then run the BBConvert application like so:

```
./BBConvert.exe "\\\\.\\pipe\\Arzn_AdDocR1_Control1" 32 1
```

This can be used as a simple way to investigate the rest of the server/client opcodes, and to perform one-off conversion requests. Note that this is not an accurate model of the real server, as the BBConvert process will usually run forever waiting for more jobs to be assigned, while with the above code, it will exit after one conversion is performed. However, it is useful for testing and learning purposes.

Another way to test the AS is by using the GUI configuration application provided by RIM. It is found in Start Menu-> Blackberry Enterprise Server->BlackBerry Server Configuration:


BlackBerry Server Configuration
BlackBerry Server Configuration Panel
Logging
Attachment Server
Test Attachment Server
...
...
Submit
Status...

Configuration Option: Test Attachment Service
Server: localhost
Server Port: 1900
Configuration Port: 1999
Output Format: Text
Commands: Document Contents
Doc Password:
Encoding: ANSII
Submit File: tings\root\Desktop\bes\bling-vulns\widthsbof.pdf
Output Folder: C:\Documents and Settings\root\Desktop\bes\c
The conversion is failed for the file.

This will actually connect to the AS on port 1900, and submit the document for conversion, mimicking the behavior of the BlackBerry messaging agent.

Finding vulnerabilities

The most obvious place to look for vulnerabilities is in the file format parsing code. With knowledge of the COM interfaces involved, RTTI information, and a bit of debugging it is relatively easy to find the interesting code. The shortest path to this is as follows:

For auditing distillers, there are only a few COM interfaces to worry about. All of the distillers are located in the following directory:

C:\Program Files\Research In Motion\BlackBerry Enterprise Server\AttachServer\BBDistiller

and they have nice names that say what they parse.

Each distiller exports two COM interfaces, which can be viewed with OLEView type lib viewer:

```
[
  odl,
  uuid(EE610E2F-F558-4BE3-ABFC-A43F389B8A77),
  helpstring("IArznLoadDistiller Interface")
]
interface IArznLoadDistiller : IUnknown {
    [helpstring("method LoadDocument")]
    HRESULT _stdcall LoadDocument(
                    [in] IUnknown* Wrapper,
                    [in] IUnknown* LoadParams,
                    [in] IUnknown* Document,
                    [in] IUnknown* Distillers);
    [helpstring("method LoadNode")]
    HRESULT _stdcall LoadNode(
                    [in] IUnknown* LoadNodeParams,
                    [out] IUnknown** Node);
};

[
  odl,
  uuid(E5D06F2A-03E3-463B-920C-47EB36613267),
  helpstring("IArznLoadDistillerRecognize Interface")
]
interface IArznLoadDistillerRecognize : IUnknown {
    [helpstring("method CouldOpenSource")]
    HRESULT _stdcall CouldOpenSource([in] IUnknown* Source);
};
```

In order to attempt to parse the file, interface users call:

IArznLoadDistiller->LoadDocument()

The first argument to this function varies, but semantically it is a wrapper for opening and reading data from the file. In the case of Office files, it will be an IStorage pointer. Some other formats use a COM class in BBAttachEngine.dll that implements ISequentialStream, which then wraps the file format so that it can be read in a similar way to an Office file. The other three arguments to this function are all COM objects from BBAttachEngine.dll. Their exact functionality hasn't been investigated.

In the actual C++ code, the LoadDocument() function is passed the IArznLoadDistiller interface pointer, and from this it obtains a pointer to the class responsible for parsing the actual document. The type of this class is dependent upon the file format, but they all use ATL and derive from a common base class. In the IDB you will find the vtables for the COM interfaces right next to the vtable for this class. Examples of this class:

ATL::CComObject<CArznFI_WordPerfect>

ATL::CComContainedObject<class CArznFI_PowerPoint>

In order to obtain these nice purty names, it is necessary to parse the RTTI data in the DLL. This can be done in IDA using the ms_rtti4.idc script from igorsk on openrce:

http://www.openrce.org/downloads/details/196/Microsoft_VC++_Reversing_Helpers

After running that script there will be tons of nice class names. The simplest way to find the vtables for all of these classes:

-load the DLL in IDA

-run the RTTI script

-Ctrl-S to jump to a segment, jump to the .rdata segment

-Alt-T for text search, search for 'IArznLoadDistillerRecognize' without the quotes.

It will be the first result, and you should see the two exported COM interfaces right next to each other. The first two vtables are for the COM interfaces. The next one is for the actual code that parses the documents (CArznFI_WordPerfect for example), which is used by the LoadDocument() function of the distiller interface. The vtable layout for the IArznLoadDistiller interface is as follows:

```
.rdata:1008B600                dd offset QueryInterface

.rdata:1008B604                dd offset AddRef

.rdata:1008B608                dd offset Release

.rdata:1008B60C                dd offset LoadDocument

.rdata:1008B610                dd offset LoadNode
```

The first 3 functions are standard IUnknown functions, and the last two are the interface for IArznLoadDistiller. Note that I have labeled those myself, there are not symbols and the RTTI scripts don't provide those names. As mentioned above, the most interesting one is LoadDocument(). This function is nearly identical for all of the distillers. The first basic block looks like this, comments identify the interesting bits.

```
; int __stdcall LoadDocument(int pIDistiller, int pStorage, int AttachClass1, int AttachClass2, 
int AttachClass3)

LoadDocument proc near
```

```
pvarg= VARIANTARG ptr -20h

lpCriticalSection= dword ptr -10h

var_C= dword ptr -0Ch

var_4= dword ptr -4

pIDistiller= dword ptr  8

pStorage= dword ptr  0Ch

AttachClass1= dword ptr  10h

AttachClass2= dword ptr  14h

AttachClass3= dword ptr  18h


mov     eax, offset sub_16DE894

call    __EH_prolog

sub     esp, 14h

push    ebx

push    esi

mov     esi, [ebp+pIDistiller]      ;interface pointer

push    edi

push    [ebp+pStorage]

mov     eax, [esi-3Ch]              ;`vftable'{for `ATL::CComObject<class CArznFI_Excel>'}

lea     edi, [esi-3Ch]              ;pointer to ATL::CComObject<class CArznFI_Excel>

mov     ecx, edi

call    dword ptr [eax+0Ch]         ;call into ATL::CComObject<class CArznFI_Excel> object

xor     ebx, ebx

mov     [ebp+pIDistiller], eax

cmp     eax, ebx

jl      loc_1685115
```

Nearly all of the work in this function is performed by virtual function calls through the file format object (class CArznFI_Excel). The order of the calls is the same for all of the different file format LoadDocument() calls, and goes like this. Note that there are a handful of basic blocks in between these calls, but that the layout and order of the calls for all the of the distillers is the exact same.

```
call    dword ptr [eax+0Ch] ; unknown
call    dword ptr [eax+20h] ; ArznFileSource(), create object to manage read file contents

call    dword ptr [eax+1Ch] ; unknown

call    dword ptr [eax+24h] ; check_file_sig(), check for magic bytes in file header usually

call    dword ptr [eax+18h] ; notImplemented1(), always seems to just return 0

call    dword ptr [eax+14h] ; parseDocument(), the bling.
```

The most interesting call is the final one, which parses the file format and sticks important data into some objects for later use by the code that builds the XML description of the file. The parseDocument() function may not directly contain the file format parsing code, but it will eventually call the function that does. It is pretty simple to find the parse loop by following through the function calls made by parseDocument(). Some file formats are very simple and you'll quickly see the parse loop, others, like Word, are complex and contain many functions that parse file code. You'll also note that a handful of distillers (image, wmf, rtf) have parseDocument() functions that just return 1/0. These filters aren't actually used. It seems as though the content is passed directly to the phone instead. This needs a bit more investigation.

The actual reading of the file data will be done through virtual function calls that usually result in ISeqentialStream calls in the case of OLE files, or the implementation of the ISequentialStream interface by some code in BBAttachEngine.dll COM classes. The logic inside of the file parsing loop is usually a big switch statement with record types as the case statements. So you'll usually see code like:

```
.text:0165F1E0 loc_165F1E0:                           ; CODE XREF: excelParseWorkbook+68□j

.text:0165F1E0                sub     ecx, excel_SST

.text:0165F1E6                jz      handle_sst

.text:0165F1EC                sub     ecx, 41h

.text:0165F1EF                jz      handle_tabid    ; 0xfc + 0x41

.text:0165F1F5                sub     ecx, 0F4h       ; 0xfc + 0x41 + 0xf4 = 0x0x231

.text:0165F1FB                jz      short handle_font

.text:0165F1FD                sub     ecx, 1EDh       ; 0xfc + 0x41 + 0xf4 + 0x1ed = 0x41e

.text:0165F203                jz      short handle_format
```

From here on the relevant data from the file is stored into various objects that are members of the core document class. So much for static analysis. If you want to fuzz, which I don't recommend, you can use the "fake-pipe" application mentioned above. Its original purpose was for fuzzing. However, I found that most of the file parsing code is rather minimal, so you'll wind up fuzzing a lot of stuff that is ignored. I had no luck fuzzing PowerPoint or Excel, but don't let that stop you.