Panther Application Security for a Desktop Application

In this document I will discuss security concerns that are specific to Panther applications, and in particular traditional Panther Desktop applications built with no more than Panther Client and a Panther Database Driver. These types of Panther applications are more difficult to harden than those using Panther Web Application Broker or a Middleware layer, since all of the critical application functionality and application artifacts are potentially accessible on the end-user's machine. The main focus will be on Panther applications running on a Windows operating system, though much of what will be discussed is applicable for other operating systems too.

A Panther application can be attacked either from within the application itself, or through modifications to the application made from outside of it. First we'll discuss internal attacks. These are possible only by means of unanticipated input that circumvents security in some way.

Internal Attack Mechanisms

The obvious internal attack mechanisms are:

A. Impersonation (user logs into the application as someone else)
B. Invalid Input (plays random havoc, and may crash the application)
C. Code Injection (clever techniques to get code entered into fields to be executed.)
D. Use of hidden features (application features or built-in Panther features that haven't been disabled)

A. Impersonation

Secure passwords are important for any application that supports different levels of program access for different users or types of users. Never store unencrypted passwords in flat files. Do not hard code unencrypted passwords in JPL files that can be accessed and decompiled by clever users. Do use a Panther password field when users need to input a password. A password field is a Single Line Text widget with its 'Password Field' property set to 'Yes'. Also, use a recent version of Panther that does not maintain unencrypted passwords in memory for password fields.

B. Invalid Input

Testing should ensure that various kinds of invalid input does not crash the application or cause it to do something it should not do. For example, make sure screens that should be opened as dialogs are not opened as sibling windows by mistake. Make sure that navigation within screens, either by keyboard or mouse, does not allow entry into widgets that should be protected under any circumstances. This is especially important to consider when there is lots of code manipulating focus and data entry protection on fields. In general, ensure that random mouse clicking on widgets and their decorations, like scroll bars, does not cause the application to do anything unexpected or even crash.

C. Code Injection

SQL Injection is a well-known hack to guard against, but the problem is not limited to SQL. Consider the following contrived JPL Screen Module code:

```
vars auth_level = 2
proc go
{
    if (1 == (:myfield))
    {
        msg emsg "all good"
    }
    msg emsg "auth_level = :auth_level"
}

proc set_auth_level(level)
{
    auth_level = level
    return auth_level
}
```

Suppose the procedure, 'go' is a control string attached to a pushbutton, and 'myfield' is a field containing some value, say 1. In that case, the default behavior of the 'go' procedure displays the message, "all good", followed by the message "auth_level = 2". Next, suppose the user types the following in 'myfield' and then presses the pushbutton:

1 && set_auth_level(1)

Now, after the "all good" message will be the message, "auth_level = 1". If auth_level is a variable containing the authorization level for the application, you can see how this exploit can be a problem. Panther's colon preprocessing feature of JPL is a powerful mechanism to exploit. It must be used only with extreme caution.

D. Use of Hidden Features

Disable hidden features that can be built into the runtime application. Do not release debuggable code to users or the PDB files that get created when building a debuggable executable. Make sure Panther's trace feature cannot be enabled by the user. (Further details on this appear later in this document.) Released applications should be built with hidden features fully disabled. For example, remove screens that are not needed from prorun5.lib, and make any others that are needed memory resident. Hard code start_screen_name in jmain.c, so that the application cannot be used to open any screen at startup.

External Attack Mechanisms

External attack mechanisms include changes to configuration variables, changes to application artifacts, replacement of application artifacts, and bit twiddling. Configuration variables can be changed in either the **SMVARS** file, **SMSETUP** file, the **OS environment**, or the application **INI** file. Application artifacts, such as screens and JPL modules can be changed or replaced. Bit twiddling can be performed on Panther application executables, meaning that bytes within the EXE file itself can be changed to bypass application security. In this section I will discuss these external attack mechanisms and how to guard against them. They will not be discussed in any particular order, as each often touches upon another in some way. Also, I will not discuss issues that would be of general concern for security of non-Panther applications too. For example, I will not discuss the prevention of external database access, which might be used to change tables and stored procedures that are used by the application.

A. Use OS Permissions

It is extremely difficult to harden a Panther application against an attack from a Panther expert. From outside of the application there are numerous ways for the expert to bypass security in order to gain complete control over the application. As a first line of defense, it is helpful to use operating system permissions to control access to sensitive application artifacts. Any artifact that does not need to have 'write' access for the user, for example, should not have such access. However, even if the user cannot change existing artifacts, copies can be made and edited. The danger is that the expert can force the application to open and execute different artifacts than those that come with the application. This can be done through environment variables that either affect the configuration directly, or point at a configuration file. Or, it may be done through such configuration variables that may be defined within the user's custom INI file.

D. Make files memory resident

As a second line of defense, screens, JPL files, reports, LDBs, video files, keyfiles, colormap files and message files can all be made memory resident. The SMVARS file can also be made memory resident. However, neither the SMSETUP file nor the INI file can be made memory resident. Therefore, for maximum security it is necessary to eliminate the possible use of the SMSETUP variable, and even to eliminate the possible use of all critical configuration variables within the INI file and the environment. Some sensitive configuration can be kept solely within a memory resident SMVARS file, while the key file, video file, message file, and colormap file can be loaded directly from memory without being specified in the memory resident SMVARS file.

Some configuration variables are an invitation to clever hackers if they are allowed to be read in by the Panther application from the environment or the INI file. For example, if one leaves in the call to sm_load_init_jpl()

within jmain.c, then any JPL modules found within the variable SMINITJPL will be loaded. Therefore, even if all JPL modules used by the application are memory resident, a name not found in the memory resident list can be used in SMINITJPL by a hacker, and then that JPL module can be an external file of unscrupulous design.

Another danger is SMTRACE. A user can normally set SMTRACE in the environment and see the internal workings of the application. The SMTRACE setting is passed to sm_trace() within the call to sm_jinitcrt() in jmain.c. The call to sm_jinitcrt() should be preceded by calls to sm_option() and sm_soption() to explicitly set any configuration variables that can be used to override the desired behavior. So, for example,

```
sm_soption (SO_TRACE, "");
```

Also, use putenv() to be certain that it won't come from the environment:

```
putenv("SMTRACE=");
```

Other variables to set include SO_SETUP, SO_LDBLIBNAME, etc. See smsetup.h.

Since the GUI INI file cannot be made memory resident, variables that can be set in the INI file can be used to override those set by sm_soption(). In order to prevent this, one may use the undocumented functions, sm_get_pi_sopts() and sm_set_pi_sopts(). Declare them like this:

```
char *    sm_get_pi_sopts PROTO((int));
int        sm_set_pi_sopts PROTO((int, char *));
```

Create a function like this:

```
static void
free_pi_sopts NOPARMS(())
{
      char *sopt;
      int i;

      for (i = 0; i < NUM_PI_OPTS; i++)
      {
            sopt = sm_get_pi_sopts(i);
            if (sopt)
                  sm_set_pi_sopts(i, (char *)0);
      }
}
```

Call free_pi_sopts() before the call to sm_jinitcrt() in jmain.c.

Also, you should do the following:

```
putenv("SMBASE=");
putenv("SMVARS=");
putenv("SMSETUP=");
putenv("SMTERM=");
putenv("TERM=");
```

As previously mentioned, the SMVARS file can be made memory resident. Use a call to sm_smsetup(char *memres_file), where memres_file is a memory resident SMVARS file. Use bin2c to create it. Place the call to sm_smsetup(memres_file) just before the call to sm_jinitcrt() in jmain.c. It is best to not have this file set SMSETUP, and to use sm_soption(SO_SETUP, "") and putenv("SMSETUP="), as well as free_pi_sopts(), just prior to your call to sm_smsetup(memres_file), in order to ensure that nothing can override the desired variables. Also, clear all other variables necessary using sm_option() and sm_soption() and putenv() prior to calling sm_smsetup(). The required variables can be found in smsetup.h, and Panther documentation should say which are settable in the environment.

Any file that Panther searches for along SMPATH that is not found in the memory resident list is a potential access point for a hacker. One can clear SMPATH using the techniques described previously, but one can still place the fake application file in the current directory for the application to find it. Therefore, such holes in the memory resident list must be closed. Only files that can do no harm, such as image files should be left in Panther libraries. If screens and JPL are left in Panther libraries, anyone with access to Panther tools like prodev, f2asc, jpl2bin, and formlib can gain sensitive knowledge from those files. Be aware that formlib's flag to configure a library for read-only access is not so much an aid to security as it is protection against accidental changes by developers to a completed library. Also, be aware that the option of jpl2bin to remove source code from the compiled JPL is merely obfuscation. JPL is an interpreted language, and much like Java class files, JPL binary-only files can be decompiled by clever means.

Obfuscation is not true security, but it can be a helpful deterrent. On that note, it would be helpful to obfuscate at least the variable names in JPL. Unlike Java class files, JPL variable names are retained in the binary-only JPL files. Prolifics does not provide a utility to obfuscate them. You could do so manually to hide sensitive variable names. Comments do get removed when source code is stripped from JPL files, so helpful comments can be retained. Note that field names are more difficult to obfuscate, because they are used in screens as well JPL.

Be aware that even if your JPL is memory-resident, someone can make a copy of your read-only executable, tweak it, and then run the tweaked executable. This is what I referred to earlier as "bit twiddling." If a hacker can locate something interesting to change in the EXE file, perhaps near to an interesting string, like "access_level", it might be possible to make an effective change to the EXE file. Individual bytes can be changed, and string constants can sometimes be changed to other string constants of the same length without compromising the integrity of the EXE file format. Or, these changes can be made in a debugger, while the unedited software is running. It isn't that difficult to do, so it is best to make it difficult to find the critical variables to change.

Instructions for making application artifacts memory resident can be found in Chapter 42, "Building Application Executables," at http://docs.prolifics.com/panther/html/dev_html/bldexec.htm. The documentation doesn't discuss it, but you may think to use a DLL to hold your memory-resident artifacts. There is no real benefit to this, and there is the danger that the DLL can be replaced by a fake one. Also, the exported symbols will help someone figure out the screen names. It is best even for that to remain difficult to figure out. If your application displays those names, because it makes it easier for developers to find the screens used by the running application, that feature is best disabled for the finished application.

Below are some issues to be aware of when using the techniques described in the Panther documentation for making your application artifacts memory resident:

Memory Resident Issues

- bin2c output does not contain filename extensions in structure names. Therefore, it can produce structures with the same names.
- The rules for Panther library entry names or file names do not match those for C variables. Therefore, a screen named "0", for example, does not produce valid output for bin2c.
- Likely increase in memory footprint of executable by several MB.
- No simple way to automatically generate form_list structure.

Glenn Silverberg

Sr Panther Developer

June 2018