# VAASeline: VNC Attack Automation Suite

Rich Smith (rich@immunityinc.com)

## Abstract

*The Remote FrameBuffer (RFB) protocol is a pervasive part of many operational networks attackers find themselves in everyday. However, until now, there has not been a scalable way to automate attacks against systems which can only be contacted over RFB. Without a way in which the human can be removed from the attack loop an attackers return on investment (ROI) is too low for the attack vector to be used in a large scale way. This paper discusses the challenges associated with trying to automate attacks only using RFB and the standard protocol messages in RFB version 3.8. A novel technique of implementing a form of Remote Procedure Calls (RPC) over RFB is presented and a library for the easy use of the techniques discussed named VAASeline will also be made available under the LGPL.*

## Background

Virtual Network Computing (VNC) allows desktops to be shared over networks and is supported by a wide variety of Operating Systems (OS) and devices. VNC is based on the Remote FrameBuffer (RFB) protocol, both created by Olivetti Research/AT&T Labs in the late 1990's [1]. The protocols are not particularly new, nor are they overtly concerned with security as while user authentication can be encrypted in most implementations such authentication is passphrase based and so vulnerable to brute force attacks. The authentication is also the only part of the RFB protocol which can employ encryption, post authentication all activity takes place via clear text. An obvious question the reader should therefore be asking is: 'Why, in 2009, would anyone expend effort on VNC compromise?', certainly a valid question but one to which the answer has more subtlety than it may first appear. The simplest answer is that there was a need to write a tool to establish a connection over RFB to a target system and take an arbitrary set of actions against that system without user involvement. While initially seeming a simple task, the more the problem was looked into the more complex it became. The more complex it became the more time was spent asking 'It really shouldn't be this difficult, should it ?'

The complexity of the problem surrounding VNC compromise is the identification or compromise of targets, but how to be able take advantage of target once compromised. RFB is a widely used protocol for both regular VNC as well as other remote control solutions that may not be explicitly labeled as VNC (For example VMWare includes RFB based management untilities). While few administrators would advocate the widespread use of VNC across their networks, the reality is that due to the ubiquity of support and sheer usefulness that VNC offers there is a very real presence of VNC/RFB in the 'Shadow IT' of production networks that attackers regularly find themselves in. This presence as a component of Shadow IT means that the already weak authentication mechanism is further weakened through lack of adherence to organisational policies for password strength, size, change period etc. If a VNC service is found to be on a system in many cases it will be one of the easiest points of entry requiring little skill on the part of the

attacker. However until now there has been no way to be able to automate post compromise actions against a VNC service using only the RFB protocol itself, it has required a human in the loop using a VNC client to make effective use of a compromise. This is not a scalable solution.
The desired solution was for an effective and reliable mechanism by which to take actions against compromised VNC services in an automated manner. To be considered successful the solution needed to satisfy a certain set of criteria, these being:

- Only use the standard RFB protocol version 3.8[2] and not any proprietary vendor extensions
- Upload an arbitrary binary to the target
- Execute arbitrary commands on the target
- Be usable and reliable over high latency, error prone links
- Initially, be usable against Win32 platforms without any extra components (bar VNC) being present
- Culminate in an extensible API which could be used in a variety of ways

The following paper details the progression in complexity of attempted techniques involved in the final solution being reached, which led to the creation of the VAASeline Python library. While specifically targeted at the RFB protocol it is felt that some of the techniques and solutions discussed could well prove useful to others dealing with automation of post compromise actions against other constrained protocols. It is not the view that VAASeline is the best, or only possible, implementation of the techniques discussed; merely that it is an example implementation upon which others can build, or find fault and improve, upon with their own implementations of the RFB control techniques discussed.

## Scalable Attack Vectors

Unlike many targets of attack, the challenges associated with attacking VNC/RFB do not centre around the initial compromise of the service, but how to make efficient and scalable use of a compromise once achieved. In general, once a system is compromised over protocol X then ideally any subsequent actions should also take place over protocol X as that path between attacker and target is established as valid.
To understand this more fully, a degree of understanding is required about the underlying RFB protocol that drives VNC and the way in which clients and servers communicate using it. The general situation can be summarised fairly simply:

RFB is a simple protocol which can be thought of as a blackbox, inputs to a server are a series of mouse updates and keystrokes, outputs from the server are a series of tile updates to a framebuffer.

# VAASeline: VNC Attack Automation Suite

Rich Smith (rich@immunityinc.com)

The protocol design is inherently geared to having a human be at the end of the connection acting as the conduit to link the keystrokes and mouse movements made, and the visual updates these cause to the visual display (the framebuffer). While well suited to it's purpose of displaying a users desktop over a network, it is unfortunately not well suited for use as an attack vector. The requirement of having a human in the protocol loop to interpret screen updates in order to determine the next course of action means the use of a VNC/RFB attack vector does not scale well. It works for simple one to one attacks, but the reduction in speed, and more importantly, return on investment (ROI) of behalf of the attacker quickly reach prohibitively low levels when it is used in a widespread manner of automated attack. The goal of the work underpinning the creation of VAASeline was to increase the attackers ROI to a point that the use of VNC/RFB as an automated attack vector became feasible, or conversely to reduce the cost of using the attack vector  down to the cost of bandwidth. Such automation would also allow defenders to include the attack vector in their auditing efforts to find, evaluate and mitigate vulnerabilities.

## The Remote FrameBuffer (RFB) Protocol

Now the motivations behind the work are clear (how to use VNC as a scalable attack vector), exactly how to go about it requires some deeper insight into the protocol which underlies VNC – the Remote FrameBuffer protocol. The protocol is surprisingly simple with a majority of the complexity tied up in efficient ways to encode and update the display (FrameBuffer) over the wire. For the purposes of attack automation, RFB protocol messages can be easily divided up into 3 distinct groups:

- Initialisation and Authentication
- Input - Client to Server messages
- Output – Server to Client messages

The first group is all to do with setting up the actual VNC connection between a client and server, it deals with agreeing protocol versions, setting the type of framebuffer and making sure the user is authorised to access the service. The second group is concerned with carrying actions from the user to the remote desktop, actions such as key presses, mouse movement and button presses and messages to allow clipboards to be kept in sync for remote copy/paste. The final group is related to what the server can present back to the client once messages from the second group have been received. This group contains only contains framebuffer updates and remote copy/paste messages, and it is this group that makes the task of automating the VNC/RFB attack vector far tougher than it would initially appear.

| Grouping for our purposes | RFB Protocol message types |
|---|---|
| Initialisation & Authentication | ProtocolVersion, Security(all), ClientInit, ServerInit, SetPixelFormat, SetEncodings, SetColourMapEntries, FramebufferUpdateRequest, FramebufferUpdate |
| Input | KeyEvent, PointerEvent, ClientCutText, |
| Output | ServerCutText, Bell |

*Table 1: The actual protocol messages split up into groups for the purposes of attack automation*

The RFB protocol simply deals with carrying input from the user to the remote system, presenting that input to the system as if the user were sitting in front of it, and then relaying the screen updates (if any) arising from that input back to the user. This can be conceptually thought of as the input group replacing the link between your mouse and keyboard, and the output group replacing the link between your monitor and graphics card. The protocol has no sense of return code, state or relation between different sets of input and output events. This means even though one can easily construct packets to simulate keystrokes, mouse movements and button presses it is difficult to write a program that is able to get meaningful output from such input as the protocol only supports the updating of a graphical framebuffer and some input may produce no update at all.

From a programmatic or packet perspective it can be seen that RFB creates a blackbox: events go in, effect the end system in some way, a set of encoded screen updates are returned. In the normal use of VNC this programmatic disconnection between input and output is rectified by the presence of a human, there is no simple way to parse the framebuffer update packets and determine the result of an input. Coming back to the earlier analogy the situation is akin to using a desktop with mouse and keyboard with the monitor turned off. Try it, high productivity is not the result.

## Simple execution

Not being able to 'see the screen' makes the use of the mouse pointer difficult to use with accuracy, one has to know where the pointer is in relation to the on-screen elements in order to interact with them. Due to this the RFB protocol message PointerEvent (essentially X, Y co-ordinates) are of little use in the pursuit of the goal of control. For this reason these protocol messages were discarded and are not discussed further. Keystrokes however can be useful, in particular hot key combinations. In RFB keystrokes are represented by KeyEvent protocol messages, with a down-flag field signifying whether the key down (1) or up(0). This means even complex key combinations and sequences are easy to create

programmatically at the protocol level. In Windows there are a variety of hotkey combinations for standard desktop operations, these can be used to achieve very simple single command execution. The following table illustrates this:

| RFB Packet sequence | Action it performs |
|---|---|
| <Windows Key Down> + <R Key Down> + <R Key Up> + <Windows Key Up> | Opens the 'Run command' window |
| <C Key Down> + <C Key Up> + <A Key Down> + <A Key Up> +<L Key Down> + <L Key Up> +<C Key Down> + <C Key Up> +<Period Key Down> + <Period Key Up> +<E Key Down> + <E Key Up> +<X Key Down> + <X Key Up> +<E Key Down> + <E Key Up> +<Enter Key Down> + <Enter Key Up> | 'calc.exe' followed by Enter |

*Table 2: RFB sequence for simple exxecution of the Windows calculator*

This will execute the windows calculator, arbitrary code execution indeed but pretty crude and not too useful! If for example the list of users on a system was wanted, the following command could be issued in the same way: cmd.exe /k net users , however the output would be unaccessible by us as it would be rendered as tile updates to the framebuffer not returned in a stdin/stdout type manner. Clearly a method by which the output of commands issued can be received is a requirement if any form of complexity is to be achieved.

## Running scripts

The next step up in complexity is only a small one but key as it acts as the bridge to the understanding of our final solution. Complex scripting in Windows can be achieved through using vbscript and the cscript interpreter, in order for this to be useful however the script needs to be transferred to target system using only the RFB protocol. For short character sequences producing a series of KeyEvent packets can work, but for longer sequences creating KeyEvent packets at machine speed this becomes unreliable. If long sequences of characters are sent to build up a vbscript a delay in between each 'keystroke' needs to be introduced in order for all packets to be received correctly and in order, remember RFB is meant to support a human typing at human speed. This slow down is obviously undesirable and the unreliability it introduces is unacceptable. In order to get around this a new technique is required to transfer strings, the remote copy/paste support built into RFB works well for this.

The ClientCutText and ServerCutText protocol messages act as a mechanism by which the contents of a clipboard can be transferred between a VNC client and server to enable copy and paste operations between them. The VNC server deals with the mechanics of receiving a ClientCutText packet and placing its content onto the clipboard, and taking any new content placed onto the clipboard and relaying this to the client as a ServerCutText packet. This mechanism is central to the proposed solution as is acting at a layer below the window manager, KeyEvent

packets are presented by the VNC server to the window manager – how those keystrokes are interpreted depends on the current state of the window manager. Client/ServerCutText packets however are received by the VNC server and copied to/from the systems clipboard buffer – this means there is no reliance on the current context of the window manager which makes this behaviour is consistent and predictable.

The use of Client/ServerCutText packets in combination with the KeyEvent packets for the 'paste' (Ctrl-V) allows the transfer of a large buffers of data to the target in a reliable way. The only limitation is that the data content of the Client/ServerCutText packets can only contain printable ASCII characters. Small scripts can therefore be executed on the target by opening the 'Run command' window via a hotkey combination and ClientCutText packets to dump “cmd /c echo 'vbscript here' > foobar” followed by an 'Enter' KeyEvent sequence. Then opening another 'Run command' window and executing “cmd /c cscript /nologo foobar ”. However the length of the command that the 'Run command' window can take is 259 characters so only the simplest scripts could be uploaded like this. This size limitation can be avoided by:

- Executing notepad with a specified filename from the 'Run Command' window,
- Dumping the vbscript to the notepad window, saving (Ctrl-S)
- Closing notepad (Alt-F4)
- and then executing cscript as before to run the script.

Difficulties here lie in knowing whether a command which spawns a window has created that window yet and where the window managers focus is. A shift in focus would mean any keystrokes being pasted by ClientCutText packets would go to a different location and our flow of control would be lost. Again the blind nature of the RFB protocol means it is very difficult to determine if/when such an error has occurred.

While certainly better than single, blind execution vbscript execution has some inherent limitations. The script needs to be constructed ahead of time to deal with any possible outcome of any included action, this means scripts can get quite large. Also in order to upload a binary through this method that binary must first be encoded to make sure it only contains printable ASCII characters (doubling its size), and transferred to a temporary file on the target, followed by a vbscript to unencode the binary. This vbscript must then be run to unencode the binary and dump it to disk, finally the binary must be executed and passed any relevant parameters. This is quite long and complex set of operations that when attempted in practice resulted in high failure rates, most often on the copy/paste transfer of an encoded binary due to its size. All but the most trivial binaries took considerable time to transfer and often were larger than notepad could deal with. This means Wordpad had to be used which introduced a whole new set of complications such as not being able to specify the filename upon Wordpad launch meaning simple Ctrl-S could not be used.

With these more complex actions each step relies on the previous step having

been completed successfully e.g. the encoded binary can only be pasted when the editor window is open and the vbscript can only be run once it has been successfully saved. The success of subsequent steps depends on the success of previous steps, and it is impossible to know if a previous step has been successful due to the reasons already discussed.
All in all while a step in the right direction, this method was to limited and error prone to be relied upon in the real world. A more robust approach was needed, ideally one in which execution state and synchronisation could be ensured.

## GuerrillaRPC

To get around the identified short comings and associated unreliabilities of having no way of knowing when a window pops or when a buffer has been fully pasted etc a more innovative approach is required. Being able to track the state of our actions and any errors that arise is key to the reliable execution of multi-step attacks. The final solution arrived at as satisfying all the prerequisite conditions was to try and implement a form (albeit a crude one!) of Remote Procedure Calls (RPC) using only the RFB protocol. The RPC approach builds upon the previous techniques discussed but attempts to maximise speed and reliability by using the Client/ServerCutText packets as much as possible for both data and control messaging. The reason why Client/ServerCutText packets significantly increase reliability is because they are intercepted by the VNC server itself and are not passed on to be action events in the window manager. This means that regardless of what is happening in the window manager if our mechanism of control is based around the Client/ServerCutText packets we do not have to wait or blindly predict when an event has happened and hope that focus has not been moved from the target window.
As previously mentioned once a VNC connection is established it is common place with VNC clients/servers that all content copied into the clipboard buffer on either side of the connection is transferred to the other VNC party via the Client/ServerCutText packets* where it is then placed in that parties clipboard buffer. This is the only truly 2 way communication channel available in the RFB protocol and so should be well suited to help achieve the final goal. The gameplan with the use of the  Client/ServerCutText packets is to use them as the I/O channel over which a process on either side of the RFB connection can use to communicate. The specifics of how to achieve this are detailed below, but the general model of operation can be summarised as:

- The client sends commands encoded inside a ClientCutText packets
- The server differentiates the 'command' clipboard content from regular clipboard content and takes appropriate action based upon the sent command
- The server gets the output/status of the command and conveys this back to the client via a response encoded in a ServerCutText packet

- The client is now able to respond accordingly, starting the protocol cycle once again

In terms of the VNC connection the attacker is the client and the target the server, implementing a process on the clientside to use our I/O channel is therefore simple as all that is needed is a specialised VNC client that concerns itself with sending, receiving and interpreting Client/ServerCutText packets and not with the visualisation of the FrameBuffer. On the target however we need to instantiate a process which will be able to get access to the contents of the clipboard buffer that the VNC server is updating with received ClientCutText packets and act upon them accordingly. It has already been discussed how to upload and execute a small vbscript on the VNC server so this seems a reasonable mechanism by which to monitor clipboard activity, recognise the presence of 'special' content, take actions based on that content and return status by placing new content into the clipboard. The VNC server will then take the task transferring this clipboard contents to the client via a ServerCutText packet, all the script has to do is get it to the clipboard. So the method of operation to achieve this on a Windows system is as follows:

- Transfer a vbscript to the target
- Run this vbscript
- Now the vbscript is monitoring the clipboard waiting for special input
- Client sends ClientCutText packets with specially formatted contents
- The VNC server receives the ClientCutText packet and places it on the clipboard
- The vbscript retrieves special input, takes appropriate action and gets output from the action
- The vbscript transfers the output to the clipboard, formatted in the correct way so as the client can interpret it
- The VNC Server then transfers this back to the client via a ServerCutText packet

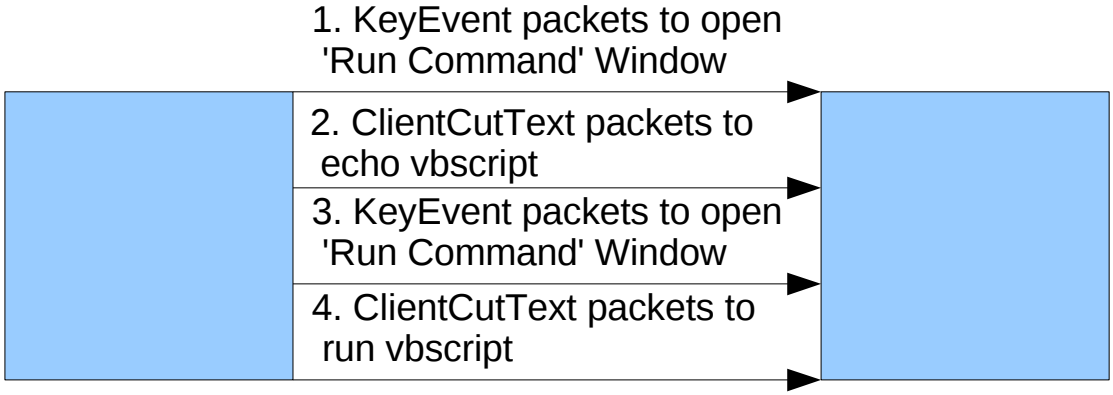


*Figure 1: First stage of the RFB RPC technique*

It is only the first 2 steps that require any use of KeyEvent packets in order to pop

the 'Run Command' window, echo a vbscript stub and execute that stub. All steps after that occur solely using Client/ServerCutText packets and so essentially happen in the background. This means once a vbscript capable of reading, writing and interpreting the clipboard contents is running on target, the attacker no longer has to worry about issues such as window focus. All ClientCutText packets sent to the VNC server will be added to the clipboard by the VNC server automatically, and all new content added to the clipboard will be returned to the attacker via a ServerCutText packet.

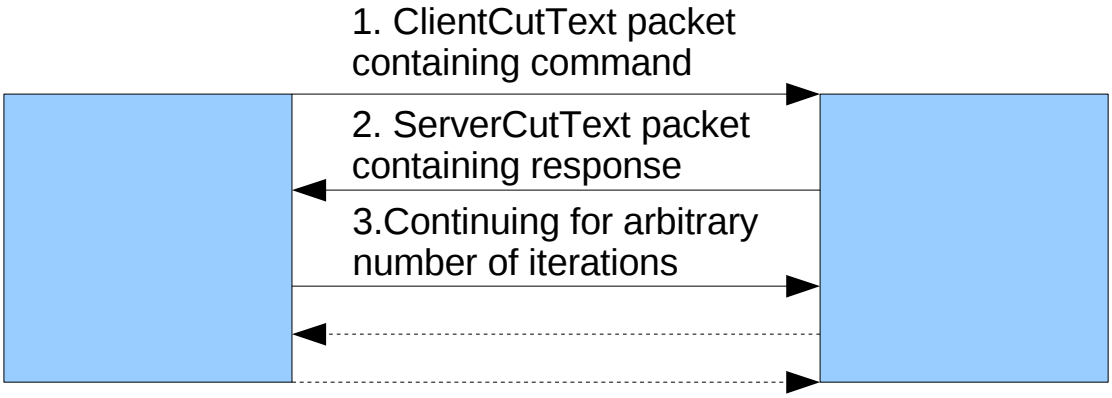


*Figure 2: Second stage of RFB RPC technique*

In this way we are increasing our reliability, speed, stealth and flexibility as much as possible. Using this workflow an arbitrarily extensible interface can be created by which to control a compromised VNC server using only the RFB protocol. An example implementation has be built using python, details of which are discussed in the next section along with the specifics around the requirements of the clipboard monitoring vbscript and steps to increase reliability as much as possible.

# VAASeline

VAASeline is a python library which implements the above workflow and exposes it via a high level API which allows actions such as initialising a clipboard monitor on the target, executing system commands, uploading binaries and getting the return status from all actions to be abstracted away from the RFB RPC method over which it is achieved.

### *The vbscript*

By default there is no way to get access to the clipboard buffer from a vbscript, indirect access can however be gained through the use of Internet Explorer. The following vbscript stub illustrates a simple loop that prints out the new content of the clipboard as it arrives:

```
'An IE object which will give access to the clipboard
Wscript.StdOut.WriteLine("Creating clipboard object")
Set objIE = CreateObject("InternetExplorer.Application")
objIE.Navigate("about:blank")

do while sitInLoop
  'Get contents of clipboard
  curr_buff=objIE.document.parentwindow.clipboardData.GetData("Text")

  If curr_buff <> prev_buff Then

    Wscript.StdOut.Write("Got new clipboard contents: ")
    Wscript.StdOut.WriteLine(curr_buff)
  wscript.sleep 1000
loop

objIE.Quit
```

Now the script is able to read the clipboard we need a way in which to signify that the contents of the clipboard should be interpreted as a command and is not just normal cut/paste activity. We also need a way to transfer define and separate commands from data over our single I/O channel.

### *VAASeline protocol*

We need to define a protocol by to communicate over our clipboard buffer I/O channel. This protocol must meet a number of constraints:

- Must be comprised entirely of printable ASCII characters – the clipboard only expects 'text' input not binary.
- NULL (ASCII 0x00) is a bad character as it terminates strings and so must be avoided
- We need a way to differentiate our input from the 'normal' contents of the clipboard buffer
- The protocol should be as simple as possible, one side of it must be completed in vbscript

The protocol used in VAASeline to meet these constraints is simple but powerful enough to meet the needs of an attacker. The protocol is made up of:

**magic (4bytes)|seqID (1byte)|opCode (1byte)|data/operands (variable)|EndOfDataMarker (1byte)**

Protocol Data Units (PDU's) are identified by a sequence of magic bytes at the beginning of the string present on the clipboard buffer. This sequence of bytes is chosen so as to minimise the possibility of false positives whereby 'normal' clipboard buffer content from copy paste operations is mistaken for a PDU being sent from the client or server. The byte sequence chose for this is

0x01,0x03,0x01,0x03 as in a vast majority of situations these character are not copied onto the clipboard. The next byte indicates the sequence ID and is used as a check between client and server that the passed messages have been received in the correct order, and more to the point are not duplicate messages. In testing it was found in some circumstances certain VNC servers would reissue ServerCutText packets multiple times when only one instance of new data had been added to the clipboard buffer. The sequence ID counter allows both client and server side to disregard duplicate PDU's with the minimum of hassle. The next byte is the OpCode, this byte determines how the remainder of the packet is interpreted by either side and the action that is taken. The opcodes for the current protocol included with VAASeline and the way the subsequent bytes are interpreted are detailed in table 3. These opcodes and actions are only an example the technique is general enough that it can be extended to take other actions as desired. Regardless of the OpCode arguments to the function defined by it are separated by start of argument and end of argument markers, these markers are 4 byte sequences of 0x02,0x02,0x03,0x03 and 0x03,0x03,0x02,0x02 respectively. If an end of argument marker is not followed by a start of argument marker all bytes between it and the end of PDU marker are considered data. The end of the PDU is signified by another magic byte sequence, this time is the single byte 0x0B.

| **Opcode** | **Operation** |
|---|---|
| 1 | echo |
| 2 | run32Cmd |
| 3 | runVBS |
| 4 | uploadExe |

*Table 3: Protocol oprcodes for the vbscript included in VAASeline*

### *Reliability measures*

While the use of the clipboard I/O channel raises reliability significantly, until this channel is set up the protocol is still at risk of failure due to the blind nature of the initial command execution. VAASeline takes a number of steps to try and increase reliability and detect errors wherever it can. Whenever a command is sent for execution prior to the clipboard I/O channel being established a best effort is made to see if the contents of the pasted buffer was transmitted correctly. This is achieved by the use of two hot key sequences: 'Ctrl-A' to select all text in the current window, and 'Ctrl-C' to copy that text to the clipboard of the target. From the attackers perspective, they know what was sent – if when the 'Ctrl-C' KeyEvent is issued the same data is returned in a ServerCutText packet then it is certain the

sent data reached the final destination. If it did not the command execution sequence (open 'Run Command' window, paste clipboard buffer) can be repeated until the correct text is returned.

The initial construction of the keyboard monitoring vbscript is also an area in which efforts can be made to increase reliability. The larger the buffer that is being transferred to the target the larger the likelihood of error in that transfer. Ideally script writing and execution should happen in a single line so as to minimise the chance of incorrect transfer through the use of the above 'select all and copy back' mechanism. For this reason the clipboard monitoring vbscript is reduced to the smallest stub possible for initial execution, the script then receives data over the clipboard buffer I/O channel and builds up its own capabilities according to the PDU's it receives. In this way arbitrarily complex RPC abilities can be created without any increase in unreliability of the initial stages as the size of the script transferred in the unreliable stage is constant.

### *Binary transfer*

The transfer and execution of binaries is handled also by the vbscript on the target. OpCode 0x04 is the opcode for upload of a binary, opcode 0x02 is then used to execute the uploaded binary. A binary is uploaded by the hex encoding of the bytes making up the binary. These are then transferred in the data portion of the PDU's and un-hex encoded by the vbscript and subsequently written to disk as a binary. The vbscript is then able to execute the binary with arbitrary sets of arguments and convey the results back to the attacker as detailed earlier. This manner of binary transfer avoids the short comings of the 'copy /paste to notepad' techniques detailed earlier and can efficiently deal with binaries of arbitrary size.

## Availability

The VAASeline library including an example and fully usable example vbscript will be available at http://www.immunityinc.com/resources-freesoftware.shtml of one embodiment of the techniques described above for the (mis)use of the RFB protocol to enable the efficient post compromise control of systems exposed over RFB, particularly systems implementing VNC on Windows.

## Futurework

The techniques detailed above illustrate a method by which post compromise actions can be efficiently
realised on Win32 platforms only. These were the obvious first target due to their global prevalence and the ubiquity of the behaviour that could e invoked by hot key sequences (the blind foothold by which all more complex actions build). The next obvious target will be the Mac OS X platforms as they too have common graphical environment hotkeys, though are less widespread. Most challenging are

the *NIX based system, most often GNU/Linux systems that can have a wide variety of window mangers and versions, with various customisations and addons residing behind a VNC server. Each will have different hot keys combinations to enable the leverage required for further actions leading to RFB RPC execution. Non RFB vectors likely are needed for use in combination with the above techniques in order to determine the underlying desktop on a target system, however the heterogeneity of the GNU/Linux environment makes the reliable use of such techniques more difficult.

## Conclusion

In conclusion it has been shown that while more difficult than one may first expect, it is possible to automate with a sufficient degree of reliability, post compromise actions on a system exposed only over the RFB protocol. The VAASeline library illustrates that the cost of attack over this vector can be reduced to the cost of bandwidth through the inventive repurposing of the RFB protocol to meet the requirements of efficient and scalable attack.

## References

1. AT&T-Cambridge original VNC website research: http://www.hep.phy.cam.ac.uk/vnc_docs/index.html
2. RFB version 3.8 protocol spec: http://www.realvnc.com/docs/rfbproto.pdf