Robert Johnson and Roy McElmurry
CSE 484
Lab 3 Report

Architecture: The architecture is designed to mimic the interface of a bash shell. Most commands take several arguments and allow for a lot of flexibility in the way that the parameters are specified. We also wanted to create a bot that was transparent to the infected user and resistant to errors on the control side. Another goal was to ensure a high level of security. For this reason crypto enjoys its own dedicated class. The bots communicate through an IRC server that acts as the hub of their communications, though at times it is necessary to circumvent the IRC server and obtain a more direct form of communication.

How it works: The slave bots join an IRC server channel and wait around for commands. Once the command and control bot joins the server, he sends an encrypted message to each of the bots present on the server. This message is encrypted with an RSA private key who's public key is hard-coded into the slave bots. The message initiates a Diffie-Hellman key exchange between the two and once finished communication can begin as expected. The CC bot may tell the slave bots to perform any of the standard actions. Leasing bots out to other CC-like bots is relatively easy. A BotnetLeaseServer must log into the IRC server and wait to obtain a lease. The CC can gather a public RSA key from the BLS bot and send this key to the slave bots. From this point forward, the slave will respond to either the CC bot or the BLS bot until the lease expires. Leasing comes with a limited set of available actions. If the CC bot controller wishes to log off of the IRC server, he may do so, but there is the option to either kill the slave bots processes or completely erase them from the computers they infect (including replacing the vulnerability). This can help to avoid legal confrontation.

Crypto: The cryptography for our Botnet was fairly widespread. We wanted to have the most privacy that we get for our bot and because of this, we used many different forms of cryptography. The communication between the CC and the bots is originally done through a hard-coded RSA public/private key pair. What we did was encrypt a message from the CC using the private key and decrypted on the bot side using the public key. The reason for this was so that our bots would know that the initial messages coming the CC are actually coming from our CC and not another bot. Thus, this would prevent man-in-the-middle attacks on start-up and when our bot leaves the network. Once the connection is verified, the CC and the bots do a Diffie-Hellman handshake to create a shared secret key between the two of them. Each bot has its own secret key with the CC. We choose this implementation so that if a bot is compromised, then the person who owned the bot will not gain any extra information on the rest of the Botnet, since they will only be able to read messages that are sent to the bot they own. Thus, each bot in our Botnet works in isolation of the other bots, and therefore not revealing any other part of the system. Once the shared secret key is obtained for the bots, they generate from it, a triple DES block cipher key that will be used between the CC and the bot. This way, only the CC and the bot should be able to communicate since only they should have the private shared secret key. We have also included MACs in the encryption as well as nonces to give extra security to our system. The MACs allow for better verification of the sender and the nonces allow for trying to prevent against repeat attacks on our system.

Threat Model: Our threat model was centered around privacy and control. We wanted to make sure that our bot was as hidden as possible, as well as secure against others as possible. We also wanted to keep as much control over our Botnet as possible, even if one or a few bots were compromised. For this

reason, we built our cryptography around utilizing all aspects of message encryption and decryption, creating ciphertexts via secret keys, using macs and nonces, and using the encrypt then mac technique for strong privacy and security of the ciphertext. Since we also wanted to have as much control as possible over our Botnet. Thus, we utilized initial cryptography so that only our CC would be able to connect to the bots. Also, by having each bot have its own shared secret key with the CC, we have reduced the chance and an attacker who gains control of one of our bots would be able to acquire any additional information about the rest of the Botnet. Another part of our threat model was detection. We did not want to be detected because our bot was running on another person's computer. Thus, we placed our bot in a hidden folder in a system folder that we thought would not be looked in. Also, we added functionality to our bot to allow for it to put the system back to before we attacked it. That way, we could leave without leaving a mark that we were on the system.

Bells and Whistles: We implemented each of the core features quite thoroughly and as such did not have much time for extra credit, but we did manage to build some bells and whistles in as we went along.

- Camouflage Your Exploit: Not only does our exploit exit normally, it also fetches what appears to be a complete html page before displaying a few strange characters and then finishing. Furthermore, the user of cgethttp will see no visual evidence that a script was downloaded and executed. We also buried the bot in a hidden folder inside the lost+found directory. It is unlikely that anyone would think twice about the "…" folder.
- Cleanup: We implemented two features that allow the control bot to be terminated quite quickly. The "kill" command simply terminates the bot process on the infected machine, this will of course start up again when the machine is rebooted. The "eradicate" command terminates the bot process and also executes a customizable cleanup script that completely erases all traces of the bot, including replacing the cgethttp vulnerability.

Part 1 Exploit Description: The cgethttp program has a simple format string vulnerability. Cgethttp prints data to the console in 2048 byte chunks via a poorly coded printf. By supplying the proper format strings we were able to work our way down the stack and overwrite the eip of one of the printing functions. The value written pointed to the buffer holding our data which then executed a call to wget followed by execution of the resulting script. Due to the relative smallness of the buffer compared to the distance between the buffer and our starting place in memory it was necessary to use direct access parameters in our format strings to access the address of the eip. Furthermore, it was important to keep the file size small because a file size under 2048 bytes but larger than the MTU 1600 could sometimes be transmitted in 2 pieces, which necessarily refilled the buffer we were trying to use for exploitation.

MD5 Hash of RandRBotnet.tar.gz:

MD5 (RandRBotnet.tar.gz) = 2431b147fdc50779922c55d72168411e

MD5(Makefile)= 6dd700aca4bf1dcbb6ca25b130dc418d
MD5(Makefile~)= 8ee86e96a407ac632af4210a319e9b15
MD5(README)= df0be9041ca4ddd26f19bb090c998e1d
MD5(README~)= ff7db02c737dd5ad45944c1c080d30fe
MD5(Report.doc)= 27bee5a1159893c11a1469eb83f33b13
MD5(asdf.txt)= 2b5539dcf8cf15dd26991ced1583fa1b
MD5(bad.sh)= f99e528b7571064c79fe10ad2850dd97
MD5(bad.sh~)= 94477305a7f72531ccc20713a212cf79

MD5(bad3.php)= 38be5929a4ee1c434cb0b82a4375aea0
MD5(bn.sh)= 74e7a1966300f23a61fe67dd2e8b24a8
MD5(bn.sh~)= 2569568294e625dfc76ee33fee9f5ceb
MD5(cgethttp.c)= 9fa2b1145da061d7455fa81fc1994e03
MD5(cgethttp.c~)= 6c1829cc76f32a0b3b438ae923df4b6d
MD5(cgethttp.h)= 49c5b8211b2498048e3606f5df63ba50
MD5(cgethttp_old.c)= 9eca2f68f3a60cd1b0e320d31412b0df
MD5(clean.sh)= 4c12a89fb66dd8ad96539a05cfaa0628
MD5(clean.sh~)= 0eb9ed41dd29f2f471c63fc57b857db9
MD5(commons-codec-1.4.jar)= 82b899580da472be37055da949b731fa
MD5(emails.txt)= f2e487cd98e97e55ac70073ac1e35337
MD5(emails.txt~)= c5a88baa71eebb41a52d001249f5e891
MD5(emails_final.txt)= 25a0c7c2a5ab7df857f0de84cc9c18f0
MD5(githelp.txt)= a8ce26e86d991016f4620a8d3222d85a
MD5(mail.jar)= ca8eb72529cd71bec83c6e419ab27290
MD5(makebots.sh)= 83024c999496ea6b5a246071c7c889f6
MD5(makebots.sh~)= dfa671804c69d11d87fda6f1ae16f339
MD5(pircbot.jar)= 28cdf066bc773d03a8d317ef1c85d055
MD5(template.txt)= 57631b7cd7a21fec76fd38a614caee92
MD5(x3.bin)= 9e7ac02099b2649bd4ce037dd3c4e76b
MD5(BotnetClient$DdosThread.class)= fb8d90896322f36b41d331fe2e83215c
MD5(BotnetClient$ProcessErrorThread.class)= fac9d5a5045d599e9eb66245c73defb5
MD5(BotnetClient$ProcessInputThread.class)= 5d417fff24201503750d0beae1123bfd
MD5(BotnetClient.class)= 498407b09f7ae755ad83d1730f55f5b4
MD5(BotnetLeaseServer$InputThread.class)= bfd13618d67018a0fa3320d7e46de684
MD5(BotnetLeaseServer.class)= b79fdabcc164946edd9c356a4e1fed13
MD5(BotnetServer.class)= 042d867c17f8fed729e79bdb5d9236b3
MD5(MsgEncrypt.class)= 5697e1da5400529b639fc95341c6d99e
MD5(PubInfo.class)= 6eaecc5683e4c8bc54aeaeaf0d03cffc
MD5(#BotnetClient.java#)= 990648e81a1f0c0ffbdfeb9872647539
MD5(BotnetClient.java)= 02ab08ec6bdd7a95c8759ce29cb4c4a4
MD5(BotnetLeaseServer.java)= e37d35eb1b5b01c2a51fdcadf4151e29
MD5(BotnetServer.java)= 34e183017bf2c9c74d04e17ec8803bfb
MD5(MsgEncrypt.class)= 8eeaf18bfa97a21e285e246eab24358f
MD5(MsgEncrypt.java)= 4ec6a86a0740ba64ae93844817445d69
MD5(PubInfo.java)= 52d8c44e6a1edc09f4043557be0587fa