

本文翻译自: <https://blog.nviso.be/2018/08/21/openssh-user-enumeration-vulnerability-a-close-look/>

介绍

OpenSSH用户枚举漏洞 (CVE-2018-15473) 是由[GitHub commit](#)公开的。

这个漏洞虽然不会生成有效用户名的列表名单，但它允许猜测用户名。

在这篇博客文章中，我们对这个漏洞进行了更深入的研究，并提出了缓解措施和监控措施。

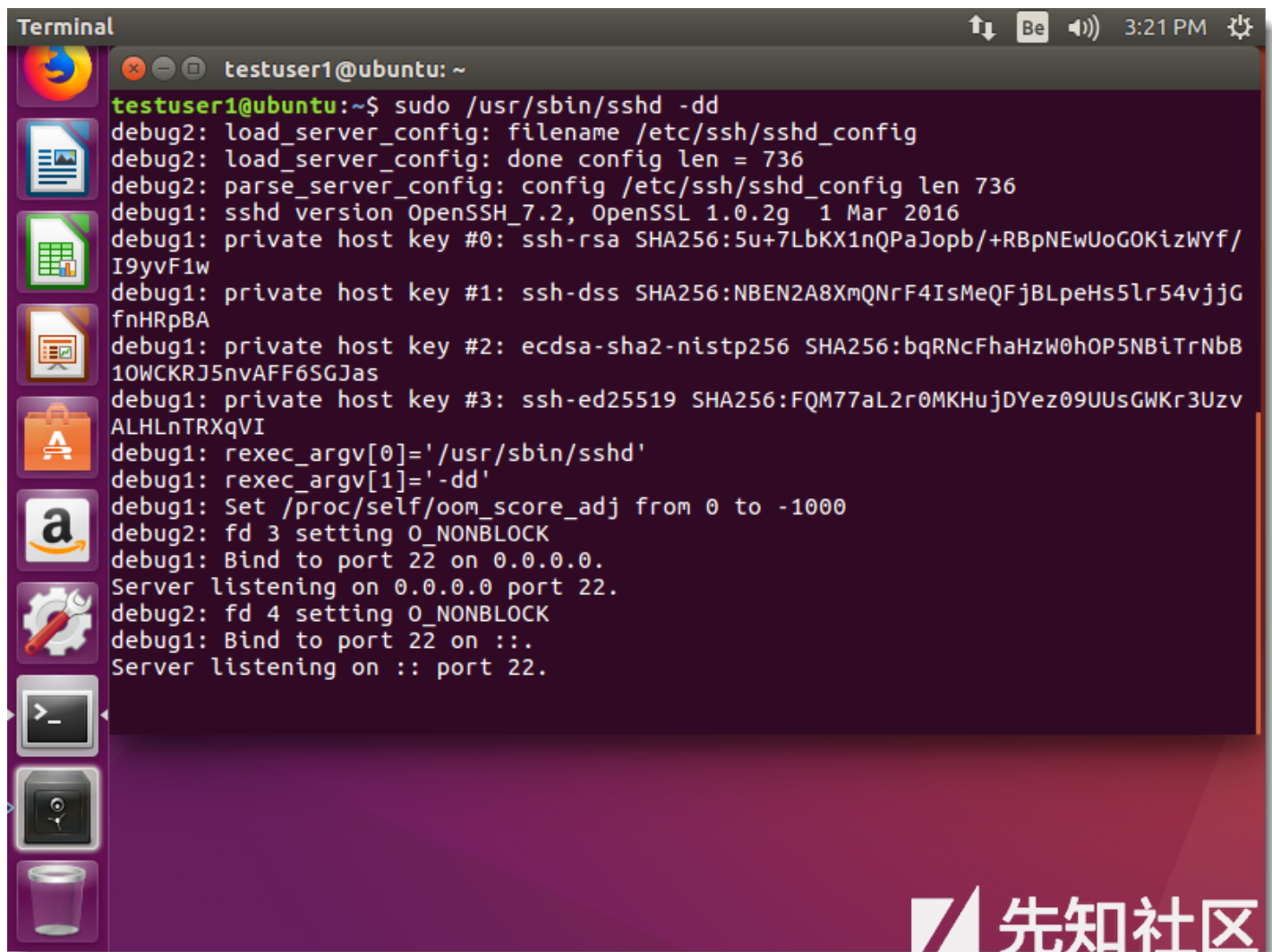
技术细节

该漏洞表现在OpenSSH的几个认证功能中。我们密切关注Ubuntu OpenSSH实现的公钥认证中的这个漏洞。

通过向OpenSSH服务器发送格式错误的公钥认证消息，可以确定特定用户名的存在。如果用户不存在，则向客户端发送认证失败消息。如果用户存在，解析消息失败将中止。

该漏洞是因为在对消息进行完全解析之前，存在用户名不存在的通信。修复漏洞本质上是简单的：逆向逻辑。首先完全解析消息，然后进行通信。

测试PoC的一种方法是在调试模式下启动OpenSSH服务器：

A terminal window titled 'Terminal' with a dark background and light text. The prompt is 'testuser1@ubuntu: ~'. The user has entered the command 'sudo /usr/sbin/sshd -dd'. The terminal displays extensive debug output from the OpenSSH server. It shows the loading of the configuration file, parsing of the configuration, and the listing of private host keys (ssh-rsa, ssh-dss, ecdsa-sha2-nistp256, and ssh-ed25519). It also shows the server setting the oom_score_adj and binding to port 22 on all interfaces. The output ends with 'Server listening on :: port 22.'.

```
testuser1@ubuntu:~$ sudo /usr/sbin/sshd -dd
debug2: load_server_config: filename /etc/ssh/sshd_config
debug2: load_server_config: done config len = 736
debug2: parse_server_config: config /etc/ssh/sshd_config len 736
debug1: sshd version OpenSSH_7.2, OpenSSL 1.0.2g 1 Mar 2016
debug1: private host key #0: ssh-rsa SHA256:5u+7LbKX1nQPajOpb/+RBpNEwUoGOKizWYf/I9yvF1w
debug1: private host key #1: ssh-dss SHA256:NBEN2A8XmQNrF4IsMeQFjBLpeHs5lr54vjgGfnHRpBA
debug1: private host key #2: ecdsa-sha2-nistp256 SHA256:bqRncFhaHzW0hOP5NBiTrNbB10WCKRJ5nvAFF6SGJas
debug1: private host key #3: ssh-ed25519 SHA256:FQM77aL2r0MKHujDYez09UUsGWKr3UzvALHLnTRXqVI
debug1: rexec_argv[0]='/usr/sbin/sshd'
debug1: rexec_argv[1]='-dd'
debug1: Set /proc/self/oom_score_adj from 0 to -1000
debug2: fd 3 setting O_NONBLOCK
debug1: Bind to port 22 on 0.0.0.0.
Server listening on 0.0.0.0 port 22.
debug2: fd 4 setting O_NONBLOCK
debug1: Bind to port 22 on ::.
Server listening on :: port 22.
```

之后，使用已存在的用户名执行PoC脚本

```
C:\Demo>ssh-check-username.py 192.168.232.193 testuser1
[+] Valid username

C:\Demo>
```

在服务器端，将会发生错误：

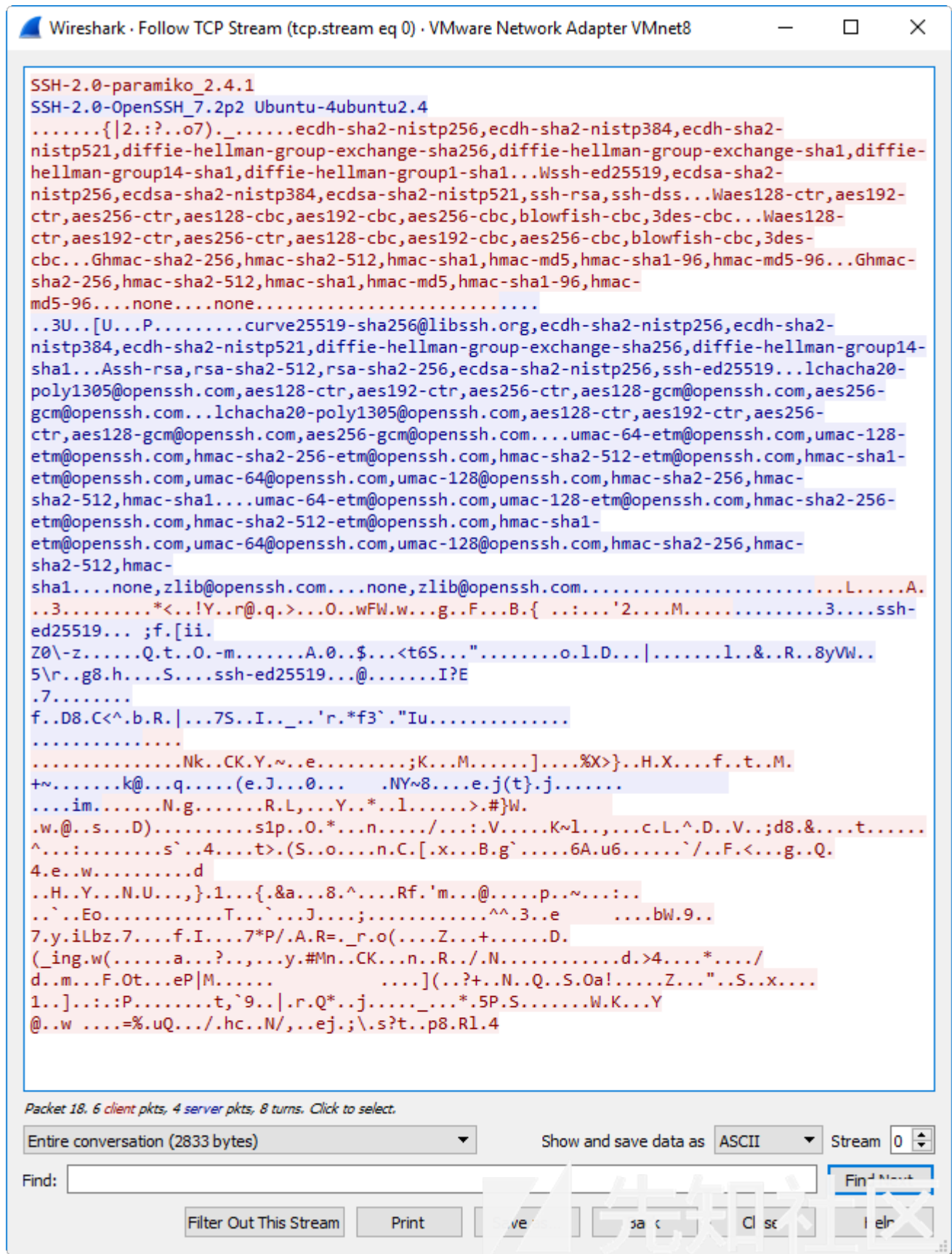
```
Terminal
testuser1@ubuntu: ~
debug2: set_newkeys: mode 0 [preauth]
debug1: rekey after 4294967296 blocks [preauth]
debug1: SSH2_MSG_NEWKEYS received [preauth]
debug1: KEX done [preauth]
debug1: userauth-request for user testuser1 service ssh-connection method public
key [preauth]
debug1: attempt 0 failures 0 [preauth]
debug2: parse_server_config: config reprocess config len 736
debug2: monitor_read: 8 used once, disabling now
debug2: input_userauth_request: setting up authctxt for testuser1 [preauth]
debug2: input_userauth_request: try method publickey [preauth]
ssh_packet_get_string: incomplete message [preauth]
debug1: do_cleanup [preauth]
debug1: PAM: initializing for "testuser1"
debug1: PAM: setting PAM_RHOST to "192.168.232.1"
debug1: PAM: setting PAM_TTY to "ssh"
debug2: monitor_read: 100 used once, disabling now
debug1: monitor_read_log: child log fd closed
debug2: monitor_read: 4 used once, disabling now
debug1: do_cleanup
debug1: PAM: cleanup
debug1: Killing privsep child 13584
debug1: audit_event: unhandled event 12
testuser1@ubuntu:~$
```

Error when parsing message

也可以在/var/log/auth.log中找到这个错误：

```
Aug 20 02:25:12 ubuntu sshd[37111]: Server listening on 0.0.0.0 port 22.
Aug 20 02:25:12 ubuntu sshd[37111]: Server listening on :: port 22.
Aug 20 02:25:16 ubuntu sshd[37112]: fatal: ssh_packet_get_string: incomplete mes
sage [preauth]
```

解析消息失败，导致客户端和服务端之间的连接关闭，而且没有来自服务器的消息：



注意，最后一个报文是粉红色的（即客户端报文），没有后续的蓝色报文（即服务器报文）。

当使用不存在的用户名来执行PoC脚本时：

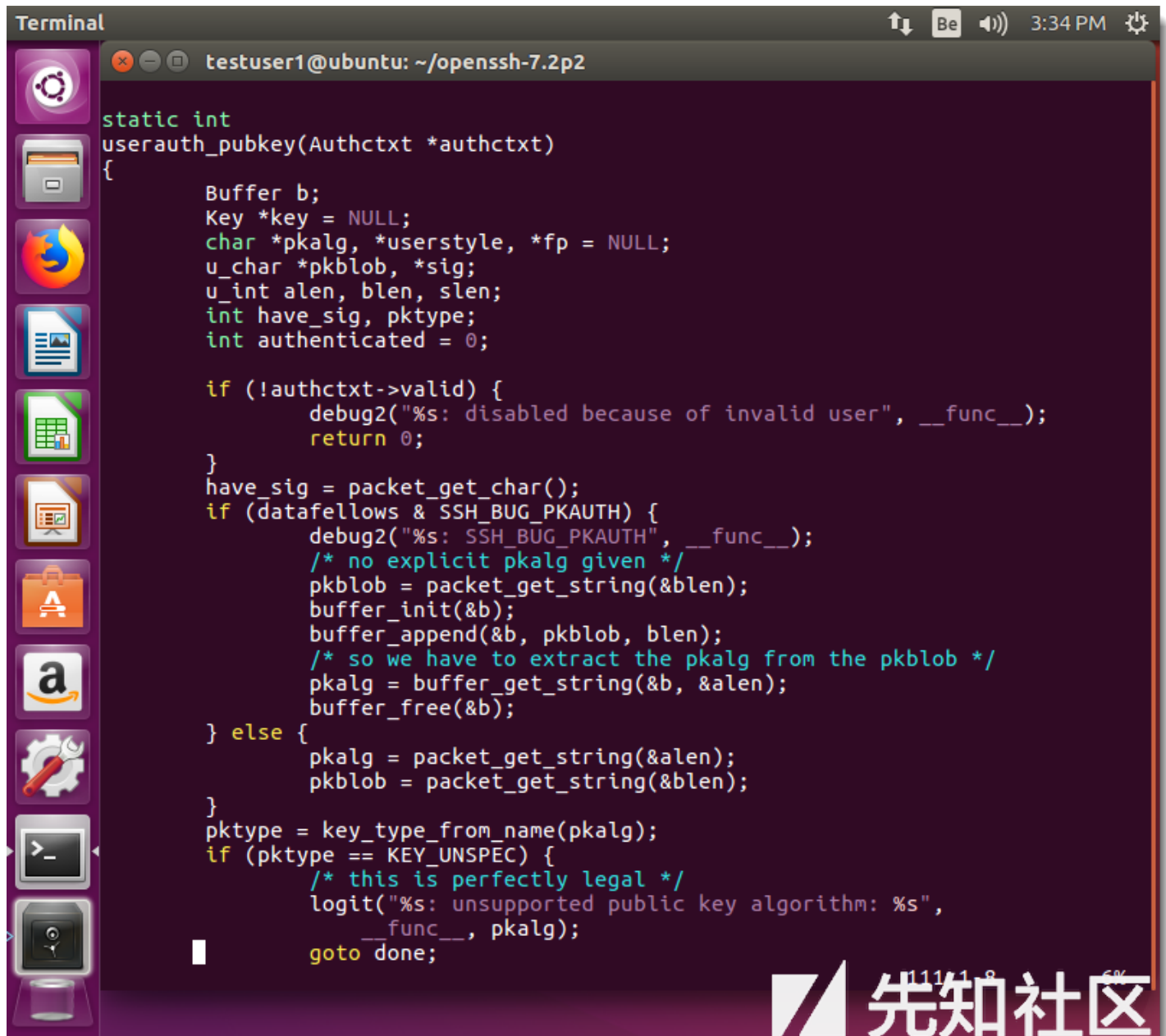
```
C:\Demo>ssh-check-username.py 192.168.232.193 testuser2
[*] Invalid username

C:\Demo>_
```

没有出现“不完整消息”的错误信息：

```
Terminal
testuser1@ubuntu: ~
debug1: SSH2_MSG_NEWKEYS received [preauth]
debug1: KEX done [preauth]
debug1: userauth-request for user testuser2 service ssh-connection method public
key [preauth]
debug1: attempt 0 failures 0 [preauth]
debug2: parse_server_config: config reprocess config len 736
Invalid user testuser2 from 192.168.232.1
debug2: monitor_read: 8 used once, disabling now
input_userauth_request: invalid user testuser2 [preauth]
debug2: input_userauth_request: try method publickey [preauth]
debug2: userauth_pubkey: disabled because of invalid user [preauth]
debug1: PAM: initializing for "testuser2"
debug1: PAM: setting PAM_RHOST to "192.168.232.1"
debug1: PAM: setting PAM_TTY to "ssh"
debug2: monitor_read: 100 used once, disabling now
debug2: monitor_read: 4 used once, disabling now
Connection closed by 192.168.232.1 port 52396 [preauth]
debug1: do_cleanup [preauth]
debug1: monitor_read_log: child log fd closed
debug1: do_cleanup
debug1: PAM: cleanup
debug1: Killing privsep child 13593
debug1: audit_event: unhandled event 12
testuser1@ubuntu:~$
```

服务器向客户端发回消息：



```
Terminal
testuser1@ubuntu: ~/openssh-7.2p2

static int
userauth_pubkey(Authctxt *authctxt)
{
    Buffer b;
    Key *key = NULL;
    char *pkalg, *userstyle, *fp = NULL;
    u_char *pkblob, *sig;
    u_int alen, blen, slen;
    int have_sig, pktype;
    int authenticated = 0;

    if (!authctxt->valid) {
        debug2("%s: disabled because of invalid user", __func__);
        return 0;
    }
    have_sig = packet_get_char();
    if (datafellows & SSH_BUG_PKAUTH) {
        debug2("%s: SSH_BUG_PKAUTH", __func__);
        /* no explicit pkalg given */
        pkblob = packet_get_string(&blen);
        buffer_init(&b);
        buffer_append(&b, pkblob, blen);
        /* so we have to extract the pkalg from the pkblob */
        pkalg = buffer_get_string(&b, &alen);
        buffer_free(&b);
    } else {
        pkalg = packet_get_string(&alen);
        pkblob = packet_get_string(&blen);
    }
    pktype = key_type_from_name(pkalg);
    if (pktype == KEY_UNSPEC) {
        /* this is perfectly legal */
        logit("%s: unsupported public key algorithm: %s",
            __func__, pkalg);
        goto done;
    }
}
```

该函数的逻辑如下：

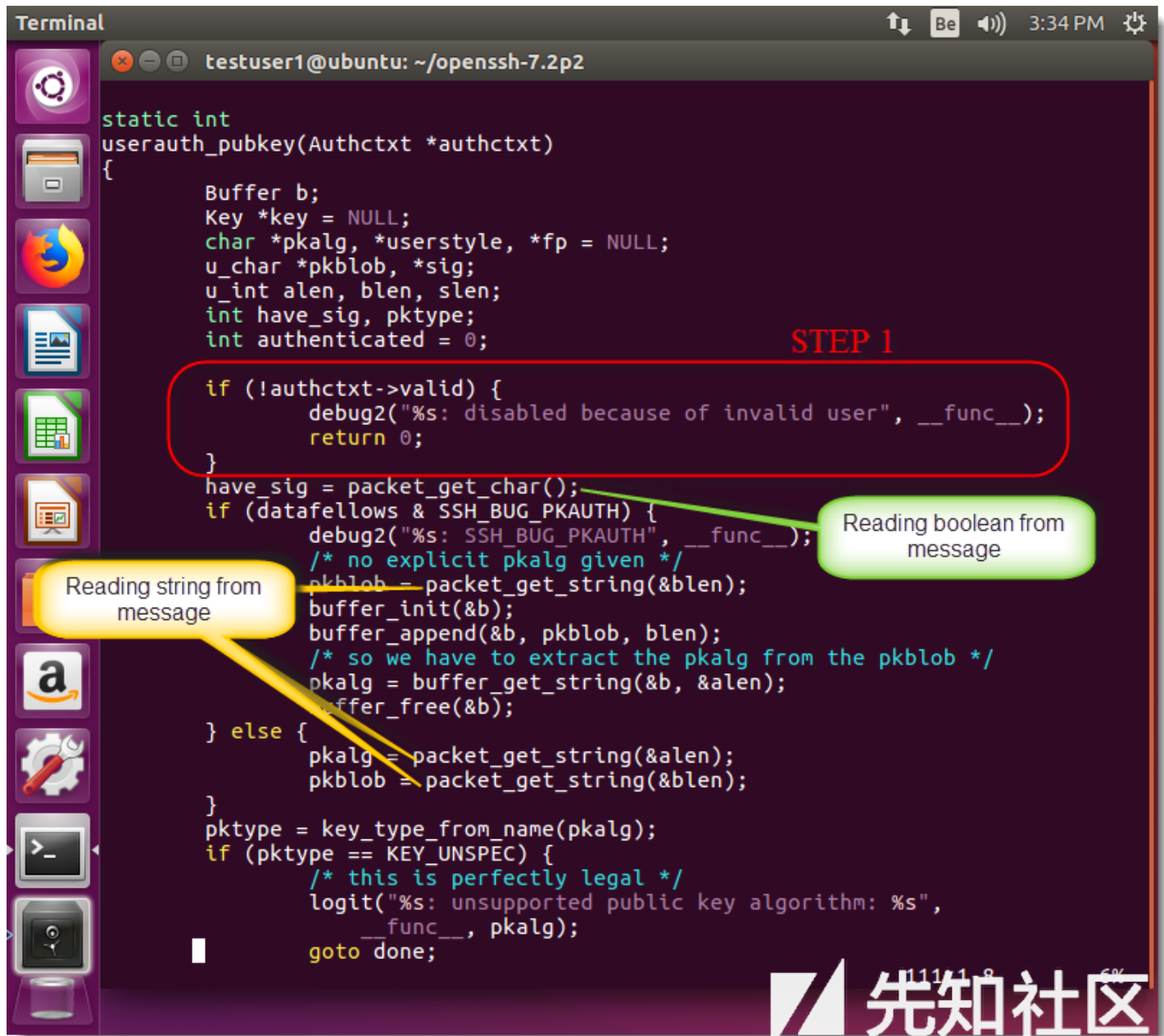
如果是未知用户名->0

如果是已知用户名，但密钥不正确->0

如果是已知用户名，并且密钥正确 ->1

一个很聪明的人想到的是，在步骤1和步骤2之间可以停止执行函数userauth_pubkey。步骤1后，函数userauth_pubkey从客户端发送的消息中检索字符串。如果这个失

这种情况可能是由函数packet_get_string导致的：



如果用户名存在，则在步骤1之后将从消息中提取字段。

第一个字段为boolean（一个字节），使用`packet_get_char`提取。当认证类型为publickey时，该字段值等于1。接下来是2个字符串：算法和密钥。在SSH消息中

函数`packet_get_string`从消息中提取并验证字符串，即检查字符串的长度是否正确。但此功能依赖于其他项：

函数`ssh_packet_get_string`的定义如下：

```
#define packet_get_string(length_ptr) \
    ssh_packet_get_string(active_state, (length_ptr))
```

函数`ssh_packet_get_string`会调用函数`sshpkt_get_string`，如果其返回值不为0，则调用函数`fatal`，而函数`fatal`会记录严重错误的事件，然后终止OpenSSH进程。

```

void *
ssh_packet_get_string(struct ssh *ssh, u_int *length_ptr)
{
    int r;
    size_t len;
    u_char *val;

    if ((r = sshpkt_get_string(ssh, &val, &len)) != 0)
        fatal("%s: %s", __func__, ssh_err(r));
    if (length_ptr != NULL)
        *length_ptr = (u_int)len;
    return val;
}

```

fatal!

先知社区

现在来看看另一个函数链：函数sshpkt_get_string调用sshbuf_get_string：

```

int
sshpkt_get_string(struct ssh *ssh, u_char **valp, size_t *lenp)
{
    return sshbuf_get_string(ssh->state->incoming_packet, valp, lenp);
}

```

先知社区

sshbuf_get_string调用sshbuf_get_string_direct：

```

int
sshbuf_get_string(struct sshbuf *buf, u_char **valp, size_t *lenp)
{
    const u_char *val;
    size_t len;
    int r;

    if (valp != NULL)
        *valp = NULL;
    if (lenp != NULL)
        *lenp = 0;
    if ((r = sshbuf_get_string_direct(buf, &val, &len)) < 0)
        return r;
    if (valp != NULL) {
        if ((*valp = malloc(len + 1)) == NULL) {
            SSHBUF_DBG(("SSH_ERR_ALLOC_FAIL"));
            return SSH_ERR_ALLOC_FAIL;
        }
        if (len != 0)
            memcpy(*valp, val, len);
        (*valp)[len] = '\0';
    }
    if (lenp != NULL)
        *lenp = len;
    return 0;
}

```

先知社区

sshbuf_get_string_direct调用sshbuf_peek_string_direct：


```

int
sshbuf_get_string_direct(struct sshbuf *buf, const u_char **valp, size_t *lenp)
{
    size_t len;
    const u_char *p;
    int r;

    if (valp != NULL)
        *valp = NULL;
    if (lenp != NULL)
        *lenp = 0;
    if ((r = sshbuf_peek_string_direct(buf, &p, &len)) < 0)
        return r;
    if (valp != NULL)
        *valp = p;
    if (lenp != NULL)
        *lenp = len;
    if (sshbuf_consume(buf, len + 4) != 0) {
        /* Shouldn't happen */
        SSHBUF_DBG(("SSH_ERR_INTERNAL_ERROR"));
        SSHBUF_ABORT();
        return SSH_ERR_INTERNAL_ERROR;
    }
    return 0;
}

```



最后，sshbuf_peek_string_direct函数实现字符串的验证：

```

int
sshbuf_peek_string_direct(const struct sshbuf *buf, const u_char **valp,
    size_t *lenp)
{
    u_int32_t len;
    const u_char *p = sshbuf_ptr(buf);

    if (valp != NULL)
        *valp = NULL;
    if (lenp != NULL)
        *lenp = 0;
    if (sshbuf_len(buf) < 4) {
        SSHBUF_DBG(("SSH_ERR_MESSAGE_INCOMPLETE"));
        return SSH_ERR_MESSAGE_INCOMPLETE;
    }
    len = PEEK_U32(p);
    if (len > SSHBUF_SIZE_MAX - 4) {
        SSHBUF_DBG(("SSH_ERR_STRING_TOO_LARGE"));
        return SSH_ERR_STRING_TOO_LARGE;
    }
    if (sshbuf_len(buf) - 4 < len) {
        SSHBUF_DBG(("SSH_ERR_MESSAGE_INCOMPLETE"));
        return SSH_ERR_MESSAGE_INCOMPLETE;
    }
    if (valp != NULL)
        *valp = p + 4;
    if (lenp != NULL)
        *lenp = len;
    return 0;
}

```

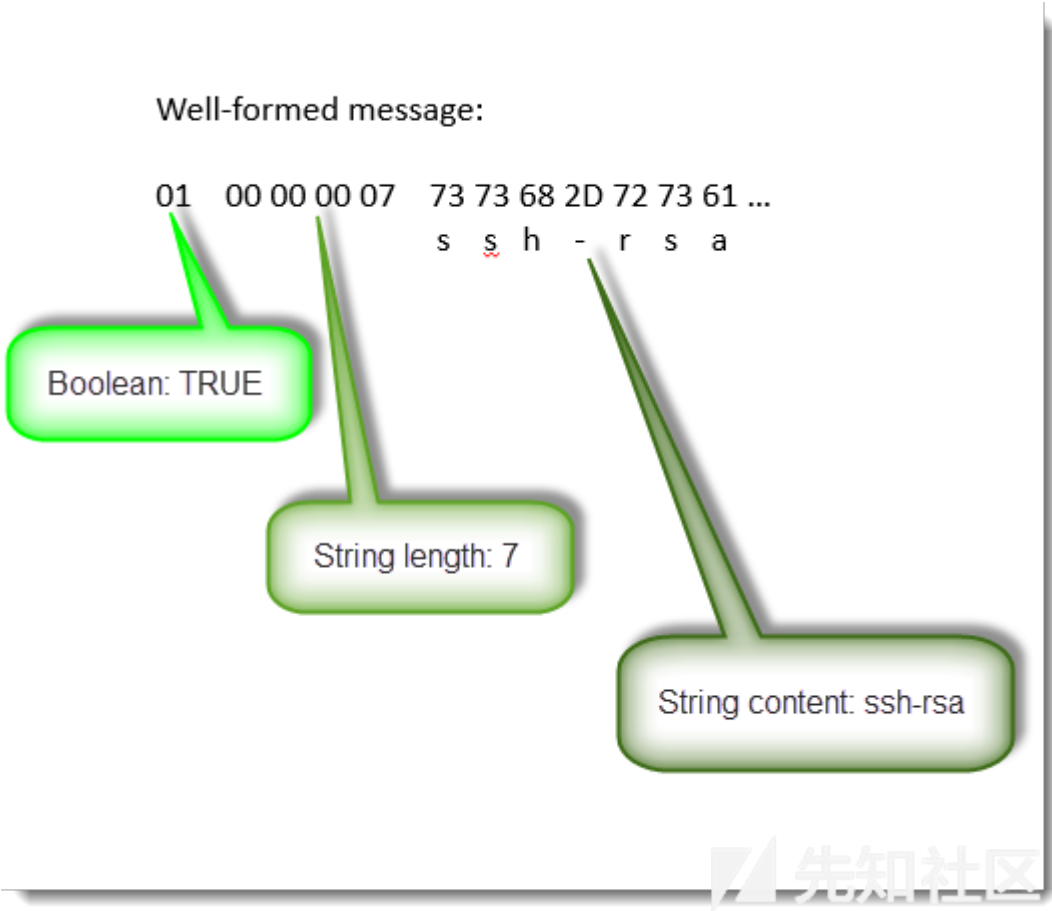


如果消息中的剩余数据小于4个字节（不能包含字符串的长度），或者消息中的剩余数据小于消息的长度，则返回错误提示信息SSH_ERR_MESSAGE_INCOMPLETE（在日志

总结这一系列函数：当 packet_get_string用于从消息中提取字符串时，如果字符串格式错误，则会发生严重的异常，从而导致OpenSSH进程终止。

这正是PoC Python脚本的触发条件。首先，它与OpenSSH服务器建立加密连接，然后发送格式错误的SSH2_MSG_USERAUTH_REQUEST（类型为publickey）消息。这个脚本将Param当函数userauth_pubkey解析这个格式错误的消息时，首先读取boolean字段。由于该字段实际上是丢失的，因此读取下一个字段的第一个字节（函数packet_get_cha

以下是格式稍好的消息的解析过程：



格式错误的消息是缺少boolean值的，但解析函数并不知道这一点，于是它将字符串的第一个字节解析为boolean字段：看起来就像消息向左移一个字节：

Malformed message:

00 00 00 07 73 73 68 2D 72 73 61 ...
s s h - r s a

Boolean: TRUE

String length: 1907

String content: sh-rsa...

先知社区

结果就是解析了1907字节长的字符串（0x00000773十六进制），这比消息本身长。因此，函数ssh_packet_get_string将调用fatal函数，导致OpenSSH进程终止。

漏洞总结

这是一个微妙的错误，它不是关于缓冲区溢出导致远程代码执行或缺少输入验证的问题。

没有缓冲区溢出，所有输入在使用前都要经过验证。这里的问题是输入验证是在一些功能处理已经发生了之后再发生的：可能出于性能原因，首先检查用户名以查看它是否存在。使用已经存在的用户名，将会进行输入验证，并且可以在不发送消息的情况下关闭连接。这可用于导出用户名的存在与否。

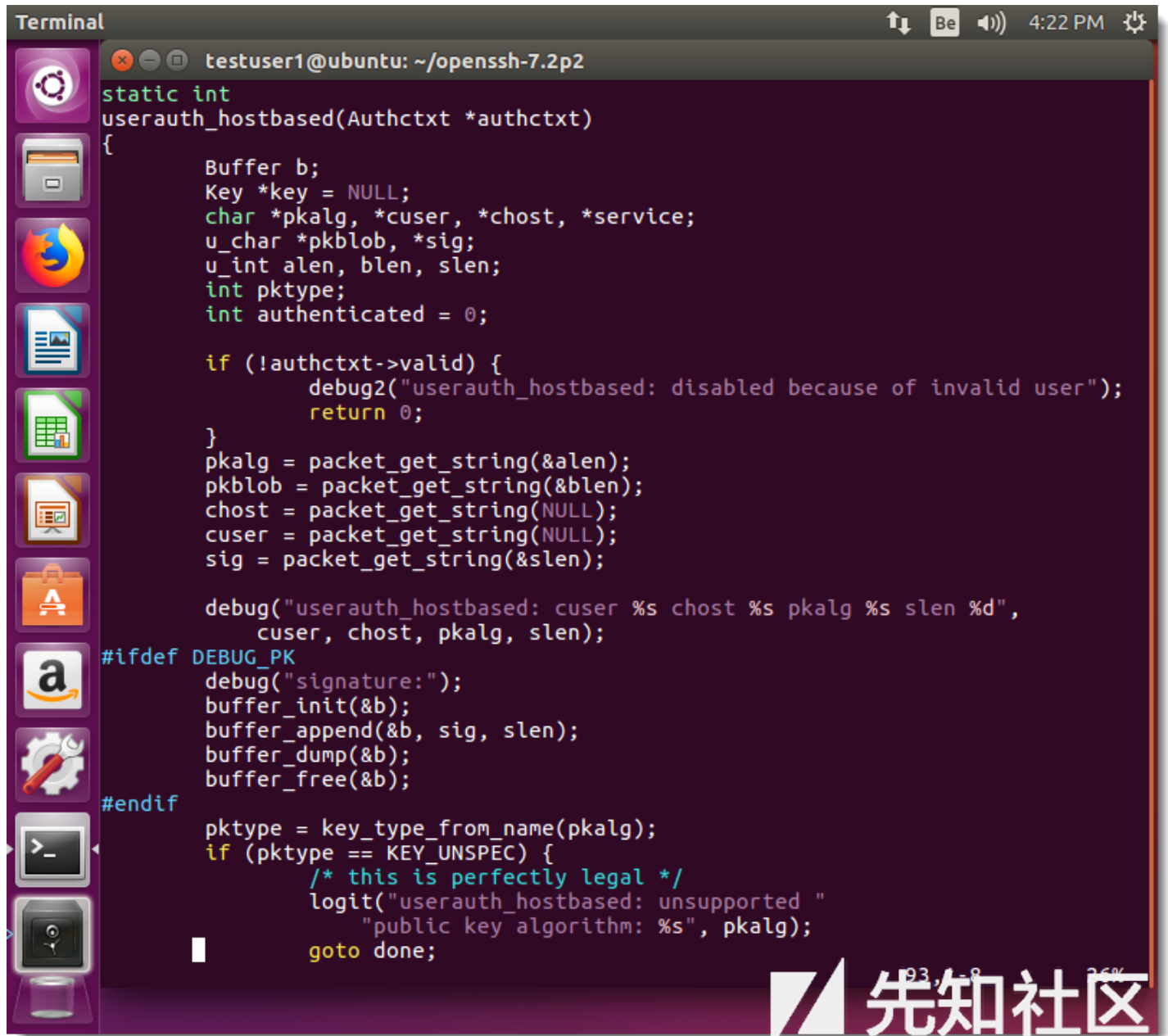
这个问题的解决方案很简单：在任何功能处理之前，切换顺序并首先进行所有的输入验证。

在其他身份验证功能中可能会出现相同的错误。一个粗略的，不完整的方法是检查表达式`authctxt->valid`，如下所示：

```
testuser1@ubuntu:~/openssh-7.2p2$ grep '!authctxt->valid' *
auth1.c:                if (!authctxt->valid && authenticated)
auth2.c:                if (!authctxt->valid && authenticated)
auth2-gss.c:            if (!authctxt->valid || authctxt->user == NULL)
auth2-hostbased.c:        if (!authctxt->valid) {
auth2-pubkey.c: if (!authctxt->valid) {
auth-bsdauth.c: if (!authctxt->valid)
auth.c:    !authctxt->valid ||
auth-pam.c:    if (!authctxt->valid || (authctxt->pw->pw_uid == 0 &&
auth-rh-rsa.c: if (!authctxt->valid || client_host_key == NULL ||
auth-rsa.c:    if (!authctxt->valid)
grep: contrib: Is a directory
grep: debian: Is a directory
monitor.c:    if (!authctxt->valid)
monitor.c:    if (!authctxt->valid)
monitor.c:    if (!authctxt->valid)
grep: openbsd-compat: Is a directory
grep: regress: Is a directory
grep: scard: Is a directory
session.c:    if (s->pw == NULL || !authctxt->valid)
testuser1@ubuntu:~/openssh-7.2p2$
```

先知社区

在基于主机的身份验证中的确犯了同样的错误（可以在GitHub [commit](#)中看到）：



```
Terminal
testuser1@ubuntu: ~/openssh-7.2p2
static int
userauth_hostbased(Authctxt *authctxt)
{
    Buffer b;
    Key *key = NULL;
    char *pkalg, *cuser, *chost, *service;
    u_char *pkblob, *sig;
    u_int alen, blen, slen;
    int pktype;
    int authenticated = 0;

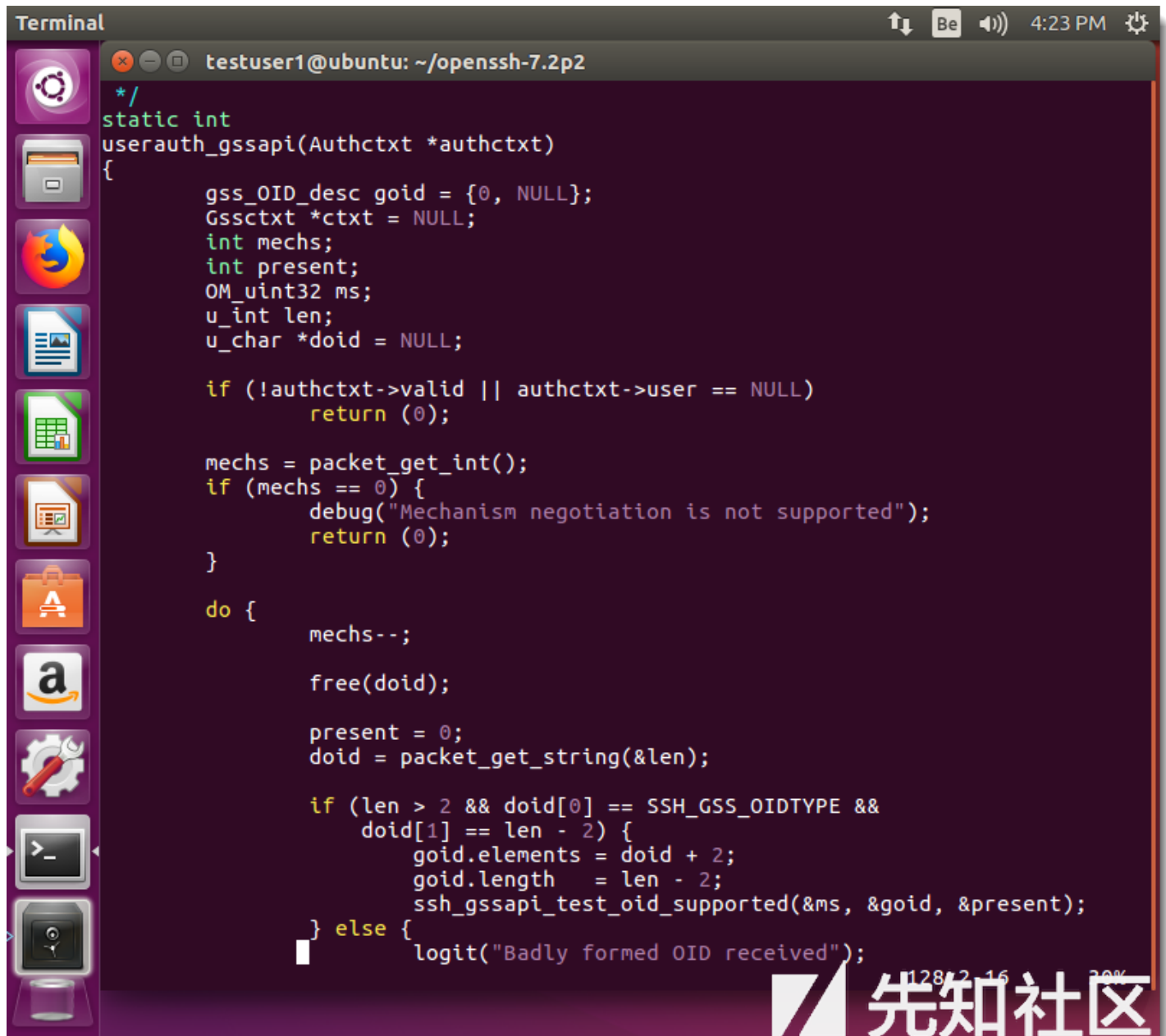
    if (!authctxt->valid) {
        debug2("userauth_hostbased: disabled because of invalid user");
        return 0;
    }
    pkalg = packet_get_string(&alen);
    pkblob = packet_get_string(&blen);
    chost = packet_get_string(NULL);
    cuser = packet_get_string(NULL);
    sig = packet_get_string(&slen);

    debug("userauth_hostbased: cuser %s chost %s pkalg %s slen %d",
        cuser, chost, pkalg, slen);

#ifdef DEBUG_PK
    debug("signature:");
    buffer_init(&b);
    buffer_append(&b, sig, slen);
    buffer_dump(&b);
    buffer_free(&b);
#endif

    pktype = key_type_from_name(pkalg);
    if (pktype == KEY_UNSPEC) {
        /* this is perfectly legal */
        logit("userauth_hostbased: unsupported "
            "public key algorithm: %s", pkalg);
        goto done;
    }
}
```

以及Kerberos的身份验证：



The image shows a terminal window titled "Terminal" with a dark purple background. The window has a title bar with standard Linux window controls and system status icons (network, volume, battery, time 4:23 PM). The terminal prompt is "testuser1@ubuntu: ~/openssh-7.2p2". The code being displayed is a C function named "userauth_gssapi" that takes an "Authctxt *authctxt" parameter. The code includes comments, variable declarations for "gss_OID_desc goid", "Gssctxt *ctxt", "int mechs", "int present", "OM_uint32 ms", "u_int len", and "u_char *doid". It contains several conditional checks and loops for negotiating GSSAPI mechanisms. The code is as follows:

```
*/
static int
userauth_gssapi(Authctxt *authctxt)
{
    gss_OID_desc goid = {0, NULL};
    Gssctxt *ctxt = NULL;
    int mechs;
    int present;
    OM_uint32 ms;
    u_int len;
    u_char *doid = NULL;

    if (!authctxt->valid || authctxt->user == NULL)
        return (0);

    mechs = packet_get_int();
    if (mechs == 0) {
        debug("Mechanism negotiation is not supported");
        return (0);
    }

    do {
        mechs--;

        free(doid);

        present = 0;
        doid = packet_get_string(&len);

        if (len > 2 && doid[0] == SSH_GSS_OIDTYPE &&
            doid[1] == len - 2) {
            goid.elements = doid + 2;
            goid.length = len - 2;
            ssh_gssapi_test_oid_supported(&ms, &goid, &present);
        } else {
            logit("Badly formed OID received");
        }
    } while (mechs > 0);
}
```

At the bottom right of the terminal window, there is a watermark logo for "先知社区" (Xianzhi Community) with the text "128/216 20%".

并且潜在的SSH1的RSA身份验证（我们没有进一步检查，因为它不再存在于OpenBSD等实现中）：



请注意，这种风险万不可轻视！

结论

您在使用OpenSSH的过程中，可以自己动手来减缓这个漏洞。在修补程序可用并部署之前，可以禁用易受攻击的身份验证机制。例如，通过禁用公钥身份验证，PoC脚本将当然，如果您不使用公钥认证，我们只建议您禁用公钥认证。如果您使用它，请不要切换到密码验证，但继续使用公钥验证！这不是远程执行代码漏洞，而是一个信息泄露漏洞。您还可以检查日志中是否有利用这个漏洞的迹象。致命错误可能是一个迹象。在Ubuntu上使用这个PoC，致命错误是“不完整的消息”。但是，此消息可能略有不同，具体得在默认配置中，您只会收到此致命错误。例如，客户端的IP地址将不会被记录。您可以通过将日志级别（LogLevel）从INFO级别提高到VERBOSE级别，从而创建额外的日志。

点击收藏 | 0 关注 | 1

[上一篇：半自动化挖掘思路——S2-057](#) [下一篇：DockerKiller：首个针对...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)